Running Category Theory and HoTT on Language: Typed Tokenization as a Software-First Path to Structured AI Reasoning

Douglas C. Youvan

doug@youvan.com

www.youvan.ai

December 4, 2025

Large language models today inhabit a world of vectors, matrices, and floating-point operations. Their hardware foundations are exquisitely tuned for dense linear algebra, not for the rich algebraic and logical structures that underlie rigorous reasoning. Proposals to make artificial intelligence "native" to category theory or homotopy type theory often begin at the wrong end of the stack, by imagining radically new chips or instruction sets. This paper argues for the opposite direction: instead of reengineering hardware to run Cat/HoTT, we should run Cat/HoTT on language. The right place to introduce higher structure is not the silicon, but the interface between text and tensors. We develop the idea of typed tokenization: a pipeline in which natural language is parsed into objects, morphisms, paths, and types before it is embedded into vectors and fed to a transformer. Category theory and HoTT are treated as semantic intermediate representations that can check, guide, and explain model behavior while leaving existing hardware untouched. We outline concrete architectures, discuss benefits for compositional generalization, explainability, and alignment, and sketch minimal experiments that could make this approach testable. The goal is a software-first path to structured AI reasoning that leverages, rather than replaces, the current compute stack.

Keywords: category theory, homotopy type theory, typed tokenization, large language models, semantic parsing, structured reasoning, proof checking, AI alignment, logostics, $q(t) \to \tau(t)$, symbolic neural integration.

A collaboration with GPT-5.1-Thinking-Extended. 52 pages. CC4.0.

Outline

1. Introduction: From Vectors to Structures Without New Silicon
1.1 The problem of structure in current LLM architectures
1.2 Hardware-centric dreams versus software-centric realism
1.3 Thesis: run Cat/HoTT on language, not on chips
1.4 Roadmap of the paper

2. Background: LLM Pipelines, Category Theory, and HoTT
2.1 How contemporary language models see text: tokenization and embeddings
2.2 Category theory as a calculus of composition
2.3 Homotopy type theory as a unifying language of types, paths, and equality
2.4 Prior attempts to integrate logic and neural networks

3. Why Not Reengineer Hardware for Cat/HoTT?
3.1 Hardware-native logic proposals and their attractions
3.2 Costs of silicon redesign: engineering, ecosystem, and inertia
3.3 Software plasticity versus hardware rigidity
3.4 Lessons from proof assistants and verified computing

4. Language as a Carrier of Categorical and Type-Theoretic Structure
4.1 Natural language as a noisy encoding of objects, morphisms, and properties
4.2 Identifying implicit diagrams in mathematical, scientific, and everyday text
4.3 From propositions to types: when statements secretly describe constructions
4.4 Limits of the analogy: where language resists strict formalization

5. Typed Tokenization: A Concrete Architectural Proposal
5.1 From subword units to semantically coherent spans
5.2 Assigning roles: objects, morphisms, predicates, paths, and universes
5.3 Partial typing and soft constraints for ambiguous or vague phrases
5.4 Representing diagrams as graphs over tokens and spans

6. Integrating the Cat/HoTT Layer with Transformer Models
6.1 Encoding types and diagram positions into token embeddings
6.2 Attention over structure-enriched sequences and graphs
6.3 Conditioning decoding on categorical and type-theoretic constraints
6.4 Hybrid modes: raw-text, structure-aware, and fully structured regimes

7. Checking and Guiding Reasoning with Cat/HoTT
7.1 Lightweight, incremental type checking during generation
7.2 Marking and verifying proof-like segments in model outputs

7.3 Using type errors and diagram failures as training signals
7.4 Combining symbolic checkers with gradient-based learning

8. Advantages Over Purely Statistical Language Pipelines
8.1 Compositional generalization and reuse of constructions
8.2 Diagrammatic and type-level explainability for humans
8.3 Alignment hooks: encoding forbidden states as impossible types
8.4 Bridging informal discourse and formal verification when needed

9. Challenges, Trade-Offs, and Failure Modes
9.1 Natural language vagueness, polysemy, and under-specification
9.2 Computational costs of Cat/HoTT parsing and checking at scale
9.3 Data requirements: corpora aligning text, types, and diagrams
9.4 Risks of overformalization and brittle behavior

10. Minimal Viable Experiments and Benchmarks
10.1 Narrow domains for initial prototypes: algebra, category theory, and simple HoTT
10.2 Synthetic data from formal systems and round-trip tests
10.3 Evaluation metrics for structured reasoning and consistency
10.4 Comparing typed-tokenization models to raw-text baselines

11. Extensions: Logostics, Alignment, and τ(t) as a Type Universe
11.1 Interpreting τ(t) as a living type universe for acceptable trajectories
11.2 Encoding ethical and physical impossibilities as type-theoretic constraints
11.3 Human–AI co-authorship of diagrams as an interface for alignment
11.4 From local type checking to global logostic guidance of q(t)

12. Conclusion: Software-First Foundations for Structured AI Reasoning
12.1 Recapitulating the case for running Cat/HoTT on language
12.2 Prospects for scaling typed tokenization in real systems
12.3 Long-term implications for AI, logic, and the philosophy of foundations
12.4 Open questions and future research directions

13. References
13.1 Category theory and compositionality in cognition and AI
13.2 Homotopy type theory and univalent foundations
13.3 Neural–symbolic integration and differentiable theorem proving
13.4 Language models, tokenization, and structure-aware architectures

1. Introduction: From Vectors to Structures Without New Silicon

Large language models are built on an almost embarrassingly simple numerical substrate: tokenize text, map tokens to vectors, push vectors through layers of matrix multiplications and nonlinearities, decode back to tokens. This pipeline has scaled astonishingly well, but its internal view of the world remains fundamentally flat. Everything that matters for meaning, reasoning, and structure must be squeezed into high-dimensional continuous vectors and learned indirectly from co-occurrence statistics.

For tasks that tolerate approximation and redundancy, this is often enough. For tasks that demand explicit structure—for example, multi-step mathematical reasoning, compositional program synthesis, or tightly constrained safety guarantees—the limits of "just vectors" become visible. Attempts to remedy this frequently jump to the other extreme: imagining hardware architectures whose primitives are not floating point operations but categorical composition, type constructors, or even homotopy-theoretic operations.

This paper proposes a different path. Instead of redesigning silicon to make category theory or homotopy type theory (HoTT) "native" at the hardware level, we treat them as semantic layers on top of language. We argue that the right place to insert richer foundations is not in the chip but in the interface that maps language into vectors: tokenization, parsing, and representation. If we can teach models to see text as already organized into objects, morphisms, types, and paths before it is embedded, then we can keep the existing hardware and still gain many of the benefits usually reserved for symbolic or proof-theoretic systems.

This introduction sets up that argument. We first describe the problem of structure in current LLM architectures, emphasizing what is missing when the only "primitive" is a real-valued vector. We then contrast hardware-centric dreams with a more pragmatic software-centric realism. From there, we articulate the central thesis of the paper—"run Cat/HoTT on language, not on chips"—and sketch the roadmap for the sections that follow.

1.1 The problem of structure in current LLM architectures

At a high level, modern transformers have two main representational moves: they embed tokens into vectors, and they use attention to mix information across positions. Both moves are extremely general. In principle, with sufficient scale, they can approximate a wide range of computations. In practice, however, they do not come with any built-in bias toward the kinds of compositional, typed structures that underlie explicit reasoning.

When a model processes a mathematical argument, a legal contract, or an informal proof sketch, it does not natively distinguish between "this phrase denotes an object," "this clause introduces a function between types," or "this step claims an equivalence." All such roles must be inferred indirectly, encoded in patterns of activations that are never exposed as first-class,

inspectable structure. Even when models appear to follow logical or mathematical rules, that behavior emerges from statistical regularities in the training data rather than from any explicit representation of morphisms, types, or proofs.

This creates several limitations. Compositional generalization becomes fragile: models struggle to reliably reassemble learned components in novel configurations. Internal consistency across long chains of reasoning is hard to maintain, because there is no explicit notion of a proof state or a diagram that must commute. And explanations are thin: it is difficult to extract from a mass of activations any clear account of how a conclusion follows from premises, beyond post hoc interpretations.

The core problem is not that vectors are "wrong" as a substrate, but that they lack exposed structure. There is no obvious place, in the standard pipeline, to say "this token participates in a functor," "this step must respect a type constraint," or "these two different-looking constructions are identified by a path." Everything is blended into a single numerical representation, and whatever structure exists is latent rather than explicit.

1.2 Hardware-centric dreams versus software-centric realism

In response to these limitations, one natural impulse is to move structure downwards: to imagine new hardware whose primitive operations are themselves categorical or type-theoretic. In such a vision, the chip might directly implement composition of morphisms, dependent function types, or homotopy-level equalities. Reasoning would then be "native" to the substrate, rather than simulated on top of linear algebra.

This is a compelling dream, but it faces several obstacles. Silicon is slow to change. Designing, fabricating, and deploying new architectures is expensive, risky, and tightly coupled to existing ecosystems of compilers, toolchains, and data centers. Once instantiated, hardware is rigid: any mistake or limitation becomes a long-lived constraint. And most importantly, it is not obvious that such specialized primitives would justify their cost, given how flexible and performant general-purpose accelerator hardware has become.

By contrast, the software stack above the hardware is plastic. Tokenizers, parsers, representational schemes, and model architectures can be iterated rapidly. New inductive biases can be introduced as pre-processing steps, auxiliary losses, or additional modules around or inside existing models. The history of verified software shows that sophisticated logics and type systems can be realized as software artifacts compiled to ordinary machines, without requiring special-purpose chips.

A software-first perspective therefore suggests a different strategy: instead of reengineering the hardware to conform to Cat/HoTT, reengineer the *representations* fed into existing hardware so that they carry categorical and type-theoretic structure from the outset. Rather than building a

"HoTT chip," we build a pipeline that interprets language through a HoTT-flavored lens, then encodes that structured view into vectors the hardware already understands.

1.3 Thesis: run Cat/HoTT on language, not on chips

The central thesis of this paper is simple to state: category theory and homotopy type theory should be treated as semantic layers on top of language, not as targets for hardware redesign.

Concretely, this means three things. First, we propose that natural language be mapped, as early as possible, into representations that identify objects, morphisms, types, and paths. A sentence is no longer just a sequence of tokens, but the surface form of an underlying diagram or proof fragment. Second, we treat Cat/HoTT primarily as *checkers and guides*: they provide constraints and structures that can be used to evaluate, shape, and explain model behavior, without dictating the low-level numerical implementation. Third, we insist that this can be done while keeping the numerical core—dense linear algebra on accelerators—essentially unchanged.

On this view, the key move is not to replace vectors with something else, but to ensure that the vectors actually *encode* explicit structure. Typed tokenization, structure-aware embeddings, and incremental type checking become the mechanisms by which categorical and homotopy-theoretic insights enter the system. The model still lives in vector space, but the way it arrives there is now mediated by a rich, formal semantic interface.

1.4 Roadmap of the paper

The rest of the paper develops this thesis in more detail. Section 2 reviews the necessary background: how current language models process text, the basic ideas of category theory and homotopy type theory, and prior work on integrating symbolic logic with neural networks. Section 3 examines proposals to reengineer hardware around logical or type-theoretic primitives and argues that, while conceptually attractive, they are pragmatically inferior to software-first alternatives.

Section 4 motivates the idea that natural language already carries latent categorical and type-theoretic structure. It shows how everyday statements, scientific explanations, and mathematical texts can be seen as informal descriptions of objects, morphisms, and types, and it also notes the limits of this analogy. Section 5 introduces the core architectural proposal: typed tokenization and the representation of sentences as diagrams over tokens and spans.

Section 6 explains how such a Cat/HoTT layer can be integrated with transformer models by enriching token embeddings and conditioning attention and decoding on structural information. Section 7 shows how categorical and type-theoretic constraints can be used to check and guide model reasoning, while Section 8 highlights the advantages of this approach over purely

statistical pipelines. Section 9 discusses challenges and failure modes, including ambiguity in language, computational overhead, data requirements, and the risk of overformalization.

Section 10 outlines minimal viable experiments and benchmarks that could make the proposal empirically testable. Section 11 sketches how these ideas connect to broader questions of alignment and guidance, including the interpretation of a living “type universe” that constrains acceptable trajectories. Section 12 concludes by summarizing the argument, reflecting on the long-term implications of running Cat/HoTT on language, and identifying open questions for future research.

2. Background: LLM Pipelines, Category Theory, and HoTT

To motivate running category theory and homotopy type theory on language rather than on chips, it is useful to recall what contemporary language models actually do with text, and what kind of structure Cat and HoTT make available. On the AI side, today’s systems operate through a highly optimized numerical pipeline: they shard text into tokens, embed these into vectors, and propagate them through transformer layers to predict the next token or fill in missing spans. On the mathematics side, category theory offers a very general calculus of composition, while HoTT reconceives logic and types in explicitly homotopical terms, treating equality as a space of paths rather than a mere proposition.

This section sketches both sides of that interface. First, it outlines how current models see text in terms of tokenization and embeddings. Second, it describes category theory as an abstract language of objects and morphisms designed to capture compositional structure. Third, it presents HoTT as a unifying framework where types, spaces, and equality are treated in a single homotopical language. Finally, it briefly surveys prior attempts to connect logic and neural networks, as a way of situating the proposal in a larger landscape of neural symbolic integration. The goal is not to be exhaustive, but to highlight the conceptual materials that the rest of the paper will recombine.

2.1 How contemporary language models see text: tokenization and embeddings

From the perspective of a large language model, text is first and foremost a sequence of discrete symbols. Before any meaning can be assigned, raw text is run through a tokenizer that converts it into a stream of tokens. In modern systems this is rarely word-level tokenization. Instead, models typically use subword schemes such as byte pair encoding, unigram language models, or variants of wordpiece tokenization. These methods split text into units that are statistically convenient: frequent substrings become tokens, rare ones are decomposed, and the overall vocabulary balances coverage with compactness.

Once tokenized, each discrete token id is mapped to a continuous vector in a learned embedding space. This embedding is where the model begins to encode notions like semantic similarity or syntactic role. Two tokens that appear in similar contexts tend to receive similar vectors; rare or highly specialized tokens may occupy distinct regions. Positional information is added either via positional embeddings or rotary schemes, so that the model can distinguish the same token in different positions.

The transformer core then processes these embedded sequences using self-attention and feedforward layers. Attention allows each position to aggregate information from other positions, computing weighted mixtures of token representations based on learned similarity scores. Stacking many such layers produces a hierarchy of representations in which each position's vector encodes information from an increasingly broad context. At the end, a projection from the hidden vector back to the vocabulary space yields a probability distribution over possible next tokens.

Crucially, throughout this process, the model has no explicit notion of variables, functions, types, proofs, or diagrams. Everything is implicit in the geometry of embedding space and the learned attention patterns. When the model appears to apply a rule of logic or algebra, it is really following a learned pattern over vectors, not manipulating a typed proof object. This fact is the starting point for our attempt to inject category theory and HoTT at the level of representation rather than at the level of hardware.

2.2 Category theory as a calculus of composition

Category theory was developed as a language for talking about structures and, more importantly, about the relationships between structures. A category consists of objects and morphisms between them, together with composition laws and identity morphisms satisfying simple axioms. The power of the framework lies not in its details but in its point of view: what matters is how things compose.

In this perspective, many familiar mathematical constructions instantiate the same pattern. Sets and functions form a category, but so do topological spaces and continuous maps, groups and homomorphisms, and so on. This unification permits a high-level calculus where one talks about products, coproducts, limits, colimits, adjunctions, and functors without committing to a specific underlying kind of object. The same compositional notions recur across fields.

For our purposes, two aspects of category theory are especially salient. First, it treats processes, transformations, and relationships as primary entities. If language contains statements about how one thing leads to or maps into another, those can be seen as informal references to morphisms. Second, it offers a disciplined way to talk about diagrams: collections of objects and morphisms arranged in patterns that express constraints like commutativity. To say that a

diagram commutes is to say that multiple paths of composition between two objects yield the same result.

This “diagrammatic” view aligns naturally with the structure of many arguments and explanations. A proof, a causal explanation, or a program can often be understood as showing that certain diagrams commute. When language is used to spell out such arguments, it is implicitly gesturing at categorical structure. The proposal in this paper is to make that structure explicit at the representational level, so that models can operate on text not just as sequences of tokens but as presentations of diagrams in some underlying category.

2.3 Homotopy type theory as a unifying language of types, paths, and equality

Homotopy type theory builds on the insights of both type theory and homotopy theory. In traditional type theory, types classify terms, and logical propositions can often be seen as types whose inhabitants are proofs. Homotopy theory, on the other hand, studies spaces up to continuous deformation, focusing on paths, loops, and higher-dimensional homotopies as central objects of interest.

HoTT brings these strands together by interpreting types as spaces and equalities as paths. In this interpretation, a proof that two terms of a type are equal is not just a yes or no fact; it is a path in the corresponding space. Moreover, there can be higher paths between paths, representing equalities between equalities, and so on. This hierarchy of identifications yields a rich structure where identity is not collapsed into a single primitive but unfolds into an infinite tower of higher homotopies.

One of the signature principles of HoTT is univalence, which, informally, states that equivalent types can be identified. Under univalence, isomorphic structures are not merely similar but can be treated as equal in a precise sense. This principle has deep consequences for how one reasons about structures, since it elevates structural equivalence to the status of identity.

For this paper, HoTT is interesting in at least two ways. First, it offers a framework in which logic, computation, and geometry are intertwined: proofs become programs, types become spaces, and equality becomes a geometric notion of path. Second, it makes explicit the idea that there may be multiple paths between two entities, corresponding to different ways of establishing a connection or equivalence. Language often reflects this: there may be several distinct arguments for the same conclusion, or several conceptual routes between two ideas. A HoTT-flavored representation can treat these different routes as distinct but related, rather than implicitly conflating them.

2.4 Prior attempts to integrate logic and neural networks

The desire to combine the pattern recognition strengths of neural networks with the explicit structure of logic is not new. Over the decades, many frameworks have been proposed under labels such as neural symbolic integration, differentiable theorem proving, and program-guided neural networks. These systems aim, in different ways, to reconcile discrete, rule-like reasoning with gradient-based learning.

Some approaches treat logic as a supervision signal. They encode logical constraints as differentiable penalties in the loss function, so that the network is nudged toward predictions that satisfy known rules. Others embed logical operators into differentiable architectures, representing conjunction, disjunction, and implication with smooth functions that approximate their truth tables. In these designs, the network's hidden states can be interpreted as soft truth values or degrees of belief in propositions.

Another line of work uses neural networks to propose candidate steps in proofs while relying on external symbolic engines to check correctness. Here, the neural model acts as a heuristic guide, navigating a enormous search space by suggesting promising inferences, while a classical theorem prover ensures that only valid steps are accepted. Related ideas appear in systems that synthesize programs with the help of neural models but verify the resulting programs with traditional tools.

These efforts demonstrate that logic and neural computation can, in principle, be combined. However, they usually treat logic as an external add-on: a collection of constraints or a separate module that interacts with the neural core. Category theory and homotopy type theory have rarely been integrated at the level of representation for general language, and the interface between natural language and these deep structural frameworks remains underexplored. The proposal developed in the rest of this paper can be seen as a specific, language-centric way of advancing neural symbolic integration: instead of bolting a logic engine onto a model, we alter how the model sees language so that logical and categorical structure are present from the very first step.

3. Why Not Reengineer Hardware for Cat/HoTT?

If category theory and homotopy type theory offer such compelling foundations for structured reasoning, it is tempting to imagine pushing them all the way down into the substrate. One might dream of a processor whose primitive operations are not floating-point multiplies and adds, but compositional steps in a category or constructions in a type theory. In such a world, proofs and programs would be executed "as themselves," without translation into vector arithmetic.

This vision is conceptually appealing. It promises to align the physical implementation of computation with the abstract structures we use to describe reasoning, potentially yielding systems whose correctness and interpretability are anchored in their hardware. However, this appeal masks substantial practical obstacles. Reengineering silicon is extraordinarily expensive and slow, the existing ecosystem is optimized around linear algebra accelerators, and once deployed, new architectures are difficult to change.

In this section, we articulate why hardware-native Cat/HoTT is, at least for now, the wrong place to start. We examine the attractions of such proposals, then detail the engineering, ecosystem, and inertia costs of silicon redesign. We contrast this rigidity with the relative plasticity of software layers above the hardware. Finally, we draw lessons from the development of proof assistants and verified computing, where sophisticated logics have been successfully implemented on top of ordinary machines rather than beneath them.

3.1 Hardware-native logic proposals and their attractions

Imagining a hardware substrate whose primitive operations are logical or categorical has a certain elegance. If the chip itself speaks the language of types, morphisms, and paths, then the gap between abstract reasoning and physical implementation seems to shrink. Instead of compiling logic into linear algebra, one could conceive of expressing computations directly in a proof-like or diagrammatic language that the hardware executes natively.

Proponents of this vision might argue that such an architecture could bring multiple benefits. First, correctness properties might be easier to establish if the instruction set is itself grounded in a well-understood formal system. Second, expressiveness could be increased if higher-level constructs, such as dependent types or homotopies, are supported directly rather than simulated through layers of encoding. Third, the very shape of computation might change: control flow, data dependencies, and resource usage could be managed in ways that reflect the compositional structure of the underlying logic.

There is also a psychological attraction. Many researchers who work with category theory or HoTT experience them as “more natural” languages for describing computation and mathematics than low-level machine code. It is flattering to imagine a machine that shares this taste: a device whose native “thoughts” are expressed in the same high-level frameworks that we use on the blackboard. This symmetry of description and implementation is aesthetically pleasing and can foster the hope that systems built this way will be more intelligible or aligned with human reasoning.

However, the attractions of hardware-native logic should be evaluated with a clear eye on the costs. It is one thing to envision a chip that composes morphisms and constructs paths; it is

another to design, fabricate, deploy, and support such a device in a world already dominated by accelerators optimized for vector operations.

3.2 Costs of silicon redesign: engineering, ecosystem, and inertia

Silicon redesign is among the most capital-intensive and time-consuming activities in contemporary technology. Designing a new processor involves specifying an instruction set, microarchitecture, memory hierarchy, interconnects, and power budget, then validating these designs through extensive simulation and physical prototyping. Fabrication requires access to advanced foundries and entails large non-recurring engineering costs. Any serious departure from standard paradigms must also contend with manufacturing constraints and the conservatism of industry roadmaps.

Even if a category-theoretic or HoTT-inspired processor could be specified, it would exist within a hostile ecosystem. Compilers, operating systems, debuggers, profilers, and monitoring tools are all currently built around standard architectures. The libraries and frameworks used for machine learning assume dense linear algebra kernels as their performance-critical inner loops. Cloud providers have already invested heavily in GPU and TPU fleets, with associated cooling, networking, and management infrastructure. Introducing a fundamentally different architecture into this environment would mean duplicating or radically modifying much of this stack.

There is also the issue of inertia. Hardware changes propagate slowly. Once a design is fabricated and deployed at scale, it is difficult to alter. Bugs and limitations must be worked around in software or lived with for years. By contrast, the fields of category theory and type theory are still evolving. New constructions, interpretations, and unifying perspectives emerge on timescales of years, not decades. Freezing a particular fragment of Cat/HoTT into silicon risks hard-coding what may turn out to be a local viewpoint into a global and inflexible substrate.

These costs do not make hardware-native logic impossible, but they do make it a poor first move. For most research questions about structured reasoning and language, it is overkill. We can gain access to many of the benefits of Cat/HoTT at the representation and algorithmic levels, without paying the enormous price of restructuring the physical substrate.

3.3 Software plasticity versus hardware rigidity

In sharp contrast to the rigidity of hardware, the software layers above the silicon are highly plastic. Tokenizers can be updated, parsers can be swapped, model architectures can be altered, and training regimes can be redesigned with relatively low friction. The same physical accelerator can be used to run many different models and frameworks simply by loading new code and weights.

This plasticity is particularly important when exploring new representational paradigms. If we want to experiment with treating language as a presentation of categorical diagrams or HoTT types, we can prototype such ideas as pre-processing steps, auxiliary modules, or fine-tuning procedures on top of existing models. If an experiment fails, we discard it and try another, without wasting years of engineering effort. If it succeeds, we can scale and refine it through incremental software releases.

Moreover, software can simulate a wide variety of logical and categorical structures on generic hardware. Proof assistants, symbolic algebra systems, and model checkers all run on ordinary processors. They demonstrate that sophisticated formal systems can be realized as software artifacts that compile down to conventional instruction sets. There is no principled barrier to implementing Cat/HoTT-based representations and checkers in software; the main challenges are design and optimization.

The key asymmetry is this: hardware commits us early and deeply, while software allows for ongoing adaptation. When a field is in flux, as the integration of language, learning, and high-level mathematics surely is, it is prudent to keep the foundational structures in a domain where they can evolve. Running Cat/HoTT on language via typed representations and software checkers respects this asymmetry. It lets us import the strengths of these frameworks without prematurely embedding them in silicon.

3.4 Lessons from proof assistants and verified computing

The history of proof assistants and verified computing offers a concrete precedent for the software-first approach. Systems such as Coq, Agda, Isabelle, and Lean implement powerful logics and type theories, many of which are intimately connected to category theory and variations of dependent type theory. Yet none of these systems rely on special-purpose hardware. They execute on standard processors, supported by conventional operating systems and compilers.

These tools demonstrate that it is possible to build robust ecosystems around high-level formal systems entirely in software. They provide interactive environments where users can construct and check proofs, synthesize programs from specifications, and verify properties of software and hardware designs. Importantly, the underlying logics have evolved over time: new features, tactics, libraries, and foundational principles have been introduced as the community learns and the theory develops. This evolution has been possible precisely because the formal systems are not frozen into hardware.

Verified computing also shows that one can create strong guarantees about correctness without changing the silicon. Compilers like CompCert, formally verified kernels, and machine-checked proofs of complex algorithms all rely on layering formal reasoning on top of ordinary machines.

The guarantees are established by aligning software and proofs, not by redesigning the physical substrate.

The proposal in this paper should be seen as an extension of this approach into the realm of language models. Instead of building a "HoTT chip," we advocate for a "HoTT lens" on language: a software layer that interprets text in categorical and type-theoretic terms, then uses that interpretation to guide and check the behavior of models running on standard accelerators. The success of proof assistants suggests that this is not only feasible but likely to be more flexible and sustainable than any attempt to encode Cat/HoTT directly in silicon.

4. Language as a Carrier of Categorical and Type-Theoretic Structure

If category theory and homotopy type theory are to be run "on language," then language must already contain, at least in latent form, the ingredients these frameworks manipulate: objects, morphisms, properties, types, paths, and equalities. This is not a claim that everyday speech is secretly written in a formal syntax, but that much of what we do with language amounts to describing entities, relations between them, and transformations that respect certain constraints.

Viewed this way, natural language is a noisy, context-dependent encoding of diagrams and constructions. When a physicist explains a derivation, a judge writes an opinion, or a teacher walks through a proof, they are not only stating propositions; they are outlining processes and relationships that could, in principle, be rendered as categorical diagrams or type-theoretic programs. The gap between the informal text and its formal counterpart is large, but structurally, many of the same patterns recur.

This section makes that claim more precise. First, it examines how natural language implicitly distinguishes objects, morphisms, and properties, even when no formal notation is used. Second, it shows how mathematical, scientific, and everyday texts can be interpreted as implicit diagrams. Third, it explains how propositions often smuggle in constructions that are naturally understood as types and programs. Finally, it acknowledges the limits of this analogy: there are regions of language where attempts at categorical or type-theoretic interpretation break down, and any architecture built on these ideas must respect those boundaries.

4.1 Natural language as a noisy encoding of objects, morphisms, and properties

In ordinary language, we constantly talk about things, how they relate, and what they are like. Nouns and noun phrases typically denote entities or collections of entities: "the electron," "this contract," "any continuous function from A to B." Verbs and verb phrases often indicate transformations or relations: "maps," "causes," "depends on," "is greater than," "sends each

element to." Adjectives and subordinate clauses introduce properties and constraints: "continuous," "differentiable," "monotone," "subject to the following conditions."

If we strip away surface variability, these elements align naturally with categorical notions. Nouns can be read as candidate objects of a category. Verbs that express structure-preserving relationships can be seen as morphisms between such objects. Phrases that express how one morphism factors through another, or how two different procedures yield the same result, correspond to compositions and equalities of morphisms. Even in non-technical discourse, patterns like "A leads to B," "B can be obtained from C by doing X," or "there are two ways to get from A to D" sketch diagrams in a loose sense.

Properties, when expressed as predicates or adjectives, suggest subobjects or characteristic morphisms. Saying "f is continuous" signals that the morphism f lies in a distinguished class, or that it factors through a subcategory of "continuous" maps. Saying "x is an element of A with property P" is equivalent to saying that x inhabits a subtype of A determined by P. Natural language thus provides a rich vocabulary for implicitly tagging objects and morphisms with additional structure.

Of course, this encoding is noisy. Words are ambiguous, context shifts quickly, and speakers are not obliged to maintain global consistency. Nevertheless, from the standpoint of representation, these regularities are valuable. They suggest that a parser or tokenizer equipped with categorical priors could often guess, with reasonable reliability, when a phrase is naming an object, declaring a morphism, or ascribing a property. That is exactly the kind of information a Cat/HoTT layer would want to extract before embedding text into vectors.

4.2 Identifying implicit diagrams in mathematical, scientific, and everyday text

Mathematical writing comes closest to literal diagrams. When an author writes "Consider the following commutative diagram" and draws objects and arrows, the mapping to category theory is explicit. But even in expository passages without pictures, the text often describes how various constructions relate: "First take the quotient of X by this equivalence relation, then apply the functor F, and finally restrict to the subspace Y." This is a sequence of morphisms composed in a specific order, with implied identities and factorizations.

Scientific explanations have similar structure. A typical narrative might say: "Exposure to condition A causes a change in variable B, which in turn modulates the response of system C. Under constraints D, the net effect is an increase in observable E." This is an informal causal diagram. Nodes correspond to variables or subsystems; arrows correspond to causal relationships or influences; conditions D restrict which paths or compositions are allowed. Process diagrams in engineering, flow charts in software design, and block diagrams in control theory are even more obviously categorical, though they are rarely described that way.

Everyday reasoning is less precise but still diagrammatic in spirit. Statements like “If you go from home to the store and then to work, it is longer than going directly from home to work” or “There are two different ways to solve this problem, but they end up in the same place” describe paths and equalities between paths in a loose sense. Even conversational explanations often take the form of “this leads to that, which then feeds back into this,” sketching cycles and feedback loops.

For a Cat/HoTT-aware representation, the key observation is that many texts can be profitably interpreted as narratives about diagrams. Identifying the nodes (objects), arrows (morphisms), and constraints (equalities, commutativity conditions, or higher homotopies) gives a structural skeleton that is largely independent of the surface phrasing. A model that sees this skeleton, rather than just a token sequence, can track consistency, compositional patterns, and missing steps in ways that current pipelines cannot.

4.3 From propositions to types: when statements secretly describe constructions

Traditional logic treats propositions as truth-bearing sentences: they are either true or false. Type theory adds another layer: propositions can be seen as types whose inhabitants are proofs. A claim like “for every x in A there exists a y in B such that P(x, y)” is not just a truth condition; it describes a construction that, given x, should produce some y with the required property. Under this view, proofs are programs, and propositions can be read as specifications for functions or other constructions.

Much of formal mathematical writing can be reinterpreted in this way. When a text asserts that “there exists a unique morphism f with such-and-such properties,” it is implicitly specifying a function that assigns to each configuration an appropriate morphism. When it states a universal property, it is describing how any competing construction must factor through the canonical one. These are naturally expressed in a type-theoretic language where types represent spaces of constructions and terms represent particular realizations.

Even outside formal mathematics, propositions often carry an implicit constructive reading. A statement like “given enough data, one can estimate the parameter θ” suggests the existence of a procedure, not just a possibility. An algorithm description is explicitly a program, but many informal claims about what “can be done” or “can always be arranged” are underspecified programs in disguise. They invite a type-theoretic treatment where each such claim corresponds to a type of admissible constructions, with different algorithms as different inhabitants.

For a language model, recognizing that certain propositions are really about constructions opens up new representational possibilities. Instead of encoding a sentence as a static assertion, the model can represent it as describing a type, possibly with parameters, whose inhabitants are the processes that satisfy the described conditions. In a HoTT-flavored setting,

these types have additional structure: they can be thought of as spaces of constructions, with paths representing continuous deformations or equivalences between different implementations. This aligns well with the idea that two different algorithms or proofs can be "the same in essence" even if they differ in details.

4.4 Limits of the analogy: where language resists strict formalization

The picture sketched so far is intentionally optimistic. It emphasizes the ways in which natural language appears to carry categorical and type-theoretic structure. However, language is not a formal system, and many uses of language actively resist strict formalization. Any attempt to build a representation pipeline based on Cat/HoTT must confront these limits.

First, ambiguity is not a bug in natural language; it is a feature. The same word can function as a noun or verb in different contexts, and even within a fixed syntactic role, its meaning can be context-dependent or deliberately vague. Metaphor, irony, and indirection rely on meanings that shift and overlap in ways that are hard to capture with sharp types and morphisms. Treating all such usages as if they denoted well-defined objects and arrows risks forcing false precision.

Second, many utterances do not describe stable structures or reproducible processes. Expressions of emotion, aesthetic judgment, or social nuance often function more as signals or invitations than as descriptions of mathematical-style relationships. A sentence like "this proof feels more natural" conveys a preference that might correlate with structural properties, but it is not itself a structural claim. Attempting to encode it directly into a categorical diagram may miss the point.

Third, even in technical domains, authors mix levels of rigor. A paper might oscillate between formal definitions and informal "handwaving," relying on the reader to fill in gaps. The same term can be used in slightly different senses in different parts of a text, and not all of these senses can be reconciled into a single tidy type. Human readers navigate these inconsistencies with a combination of background knowledge, charity, and pragmatic reasoning. A strictly formal system would either reject such texts as ill-typed or silently misinterpret them.

Finally, there are deep philosophical questions about the extent to which all of meaning can be captured in any single formal framework. Category theory and HoTT provide powerful lenses, but they are still lenses: they highlight certain aspects of structure while downplaying others. Using them as the organizing principle for language representation is a choice, not an inevitability.

For the purposes of this paper, these limits are not reasons to abandon the idea of running Cat/HoTT on language, but they do impose constraints on how it should be done. A practical system must allow for partial typing, soft constraints, and fallbacks to unstructured

representations. It must be willing to say “this passage does not fit neatly into a diagram” and avoid overcommitment. The next sections develop such a system in architectural terms, with the understanding that the categorical and type-theoretic view is an overlay on language, not a complete capture of it.

5. Typed Tokenization: A Concrete Architectural Proposal

The central move in this proposal is to change what “a token” is. In current systems, a token is primarily a statistical unit: a subword piece chosen to balance vocabulary size and coverage. Its semantics are encoded only implicitly through its embedding and the contexts it appears in. In a Cat/HoTT-aware system, a token (or, more precisely, a token span) becomes a carrier of explicit semantic role and type information.

Typed tokenization does not discard subword tokenization but layers a second, semantically oriented segmentation and labeling pass on top of it. The idea is to identify spans that function as objects, morphisms, predicates, or higher-structure markers; to assign them roles and tentative types; and to organize them into diagrams before they are embedded into vectors. This does not require full formalization of the text. Instead, it aims for a partial, approximate structural annotation that is good enough to guide the model and supply a backbone of explicit relationships.

This section lays out a concrete architectural sketch. First, it describes how to move from raw subword units to semantically coherent spans. Second, it explains how to assign roles such as objects, morphisms, predicates, paths, and universes to those spans. Third, it argues for partial typing and soft constraints as the right way to handle ambiguity and vagueness. Finally, it shows how to build diagram-like graph structures over tokens and spans, which can then be fed to or co-processed with a transformer.

5.1 From subword units to semantically coherent spans

Modern tokenizers are optimized for compression, not semantics. Byte pair encoding and its relatives break text at statistically convenient boundaries, producing units like “cont”, “inu”, “ous”. These are efficient for training but rarely align with meaningful entities such as variable names, function symbols, or domain-specific technical terms. Typed tokenization begins by accepting this reality and adding a second segmentation layer that operates on top of the raw token stream.

The first step is span detection. A lightweight parser or learned tagger groups contiguous tokens into spans that are likely to function as coherent semantic units: “continuous function from A to B,” “for every element x of the set S,” “the derivative of f at zero.” These spans may overlap and

nest. For instance, within “continuous function from A to B,” the phrases “continuous function,” “A,” and “B” are themselves meaningful spans. The segmentation module therefore produces a lattice of spans rather than a single flat partition.

This segmentation can be informed by a mix of signals. Syntactic cues from part-of-speech tagging, dependency parsing, or chunking can identify noun phrases, verb phrases, and prepositional phrases. Domain-specific recognizers can spot common patterns like “from X to Y,” “subset of,” or “equivalence relation on Z.” Prior knowledge about mathematical or scientific notation can guide the detection of variable names, function symbols, and type-like expressions. Over time, the system can learn from data which spans tend to play which roles.

Crucially, this segmentation is not required to be perfect. The goal is to provide candidate chunks that downstream modules can interpret and refine, not to enforce a single correct bracketing. It is better to over-generate plausible spans and later prune or disambiguate them than to miss important semantic units. Typed tokenization thus starts from the existing subword stream but quickly lifts the representation to a richer, span-based view in which potential objects and morphisms can be identified.

5.2 Assigning roles: objects, morphisms, predicates, paths, and universes

Once spans have been identified, the next step is to assign them semantic roles. In a Cat/HoTT-aware system, the primary roles might include at least the following:

- Object-like roles: spans that denote entities, sets, types, spaces, or structures.
- Morphism-like roles: spans that denote functions, maps, transformations, implications, or causal links.
- Predicate-like roles: spans that express properties or conditions that can be applied to objects or morphisms.
- Path-like roles: spans that express equalities, equivalences, isomorphisms, or “up to” relations.
- Universe-like roles: spans that indicate levels of abstraction or domains of discourse, such as “the category of sets” or “the universe of all types under consideration.”

A role assignment module, which can itself be an LLM-like model or a smaller classifier, predicts for each span a role distribution. For example, “continuous function from A to B” might be tagged as morphism-like, “A” and “B” as object-like, “continuous” as predicate-like, and “from A to B” as further evidence that the main span describes a morphism $A \to B$. The phrase “is isomorphic to” might be tagged as path-like, signaling that what follows is an equivalence rather than an arbitrary relation.

These roles can be encoded as discrete tags or as vectors in a “role space” that is concatenated with or projected into the token or span embeddings. The important point is that the model now has access to an explicit suggestion about how each span should be interpreted structurally. It is not forced to infer purely from context whether a phrase names an object, denotes a map, or states a property; that information is provided as an additional channel.

For universes, the system can maintain a registry of currently active categories, type universes, or domains of discourse as they appear in the text. Phrases like “in the category of groups” or “working in homotopy type theory” can shift the active universe context. This context can then influence how subsequent spans are typed. Saying “morphism” inside one universe might mean “group homomorphism,” while in another it might mean “continuous map of spaces.”

Role assignment is not expected to be perfect or unique. Many spans may plausibly play multiple roles, and the system must allow for uncertainty. Still, even a noisy estimate of roles can significantly constrain the space of possible diagrams the text can be mapped into, making subsequent Cat/HoTT processing more tractable.

5.3 Partial typing and soft constraints for ambiguous or vague phrases

Natural language is full of ambiguity and underspecification. Typed tokenization must reflect this by adopting partial typing and soft constraints rather than insisting on full, precise types for every span. The goal is to approximate the type structure where possible while gracefully degrading to weaker information when necessary.

In practice, this means that for each span, the system may assign:

- A set of candidate types with associated confidences.
- Constraints on relationships between types rather than explicit type identities (for example, “this object’s type must be in the currently active universe,” or “this morphism’s domain must match the type of span X”).
- Generic placeholders for unknown or only loosely constrained types, such as “some type in this universe” or “some object related by a specified property.”

When the text is precise, such as in formal mathematics or code, these candidate types may collapse to a single, well-specified type. When the text is informal or vague, the system may only be able to assign a coarse type, such as “some kind of object in this domain” or “a relation between two things of similar type.” The type system must be able to carry such partial information forward and refine it as more context arrives.

Soft constraints come into play when checking or using this type information. Rather than treating a mismatch as a hard error that invalidates a representation, the system can assign

penalties to type-inconsistent interpretations. During training, it can backpropagate from these penalties, nudging the model toward representations that satisfy more constraints. During inference, it can downweight candidate continuations that would dramatically violate the current typing or role assignments.

This approach also allows the system to cope with mixed registers. Within a single text, some segments may be amenable to strong typing and diagrammatic interpretation, while others are primarily narrative or rhetorical. Typed tokenization does not have to force everything into Cat/HoTT; instead, it treats structure as an overlay where applicable. Spans that resist typing can be marked as untyped or weakly typed and passed through to the vector model with minimal structural constraints.

In essence, partial typing acknowledges that language and logic interact on a spectrum. Typed tokenization aims to capture the parts of the spectrum where structural information is both present and useful, without overreaching into regions where formalization would be artificial or misleading.

5.4 Representing diagrams as graphs over tokens and spans

With spans identified, roles assigned, and partial types in place, the final component of typed tokenization is to assemble these pieces into explicit diagram-like structures. These diagrams are graphs whose nodes represent spans (or aggregated entities derived from spans) and whose edges represent candidate morphisms, constraints, and higher relationships suggested by the text.

A simple diagram might have nodes for object-like spans and edges for morphism-like spans, with edges labeled by the corresponding function names or relational descriptors. Path-like spans, such as “is equal to,” “is isomorphic to,” or “commutes with,” can introduce higher-level relationships: equivalence relations between nodes, identifications of parallel paths, or commutative square constraints. Predicate-like spans attach labels to nodes or edges, tagging them with properties or indicating that they belong to specific subcategories.

These graphs need not be strictly categorical in the formal sense; rather, they are candidate diagrams that approximate the structures a human reader might infer. Some edges may be underspecified; some nodes may serve as placeholders for implicit objects. Importantly, the graphs can be dynamic: as more of the text is processed, new nodes and edges can be added, and existing structures can be refined or merged.

From the perspective of a transformer model, these diagrams can be integrated in several ways. One approach is to augment each token or span embedding with features derived from its position in the graph: node degree, role, type summary, and information about adjacent nodes and edges. Another is to run a separate graph neural network over the diagram and fuse its

outputs with the sequence model's representations. A third is to let attention operate not only over the linear token sequence but also over graph edges, treating the diagram as a second connectivity structure that informs which positions attend to which.

The key advantage of having explicit diagrams is that they provide a substrate for Cat/HoTT-style checking and guidance. One can attempt to enforce simple categorical identities, such as the associativity of composition, within the diagram. One can detect obvious anomalies, like a morphism whose domain does not match its source node's type, and feed these as signals to the model. One can compare different diagrams extracted from different parts of a text to see whether they are consistent, thereby supporting coherence checks over long contexts.

In summary, typed tokenization transforms a flat stream of subword tokens into a richly annotated structure: a lattice of spans, each with a tentative role and type, arranged into diagrams that reflect the relationships implicit in the text. These diagrams do not replace embeddings and attention; they inform them. The Cat/HoTT layer lives here, where language is first interpreted as a carrier of objects, morphisms, and types, before everything is projected into vector space.

6. Integrating the Cat/HoTT Layer with Transformer Models

Typed tokenization and diagram extraction change what we feed into a model, but they do not require us to abandon the transformer architecture. Instead, the Cat/HoTT layer sits between raw text and the transformer, enriching the representation with structural information that the transformer can exploit. On the way out, the same layer can constrain and interpret model outputs in light of categorical and type-theoretic structure.

The integration problem therefore has two sides. On the input side, we must encode types, roles, and diagram positions into embeddings that remain compatible with dense linear algebra. On the internal side, we must adjust attention and intermediate computations so that they can take advantage of graph structure, not just linear token order. On the output side, we need mechanisms for conditioning decoding on categorical and type-theoretic constraints, without turning generation into an intractable symbolic search. Finally, we must accept that not all workloads require or tolerate the same level of structure, and design hybrid modes that range from raw-text operation to fully structured reasoning.

This section sketches how these integrations can be realized. It begins with encoding types and diagram positions into token embeddings. It then describes attention mechanisms that operate over both sequences and graphs. Next, it explains how decoding can be conditioned on Cat/HoTT constraints in a soft, tractable way. It closes by outlining a set of hybrid modes that allow systems to vary their reliance on structure depending on the task and context.

6.1 Encoding types and diagram positions into token embeddings

The simplest way to expose Cat/HoTT information to a transformer is to embed it alongside the usual lexical and positional information. Each token (or span) ultimately becomes a vector that is fed into the model. Typed tokenization gives us, for each such element, additional attributes: role tags (object, morphism, predicate, path, universe), candidate type summaries, and positions in one or more diagrams.

Concretely, we can represent the structural attributes using learned embeddings and concatenation or addition. For example, we might maintain:

- A role embedding table, mapping each role to a vector.
- A type summary embedding, produced by pooling over candidate types and their confidences.
- A diagram-position embedding, which encodes properties such as node identifier, local degree, or a hash of the neighborhood structure.

Given a base lexical embedding for the token, we can then form a composite representation by concatenating these vectors followed by a projection back to the model's hidden dimension, or by summing them after projecting them into a shared space. Positional information along the sequence is handled as usual, so the model still knows where in the sentence each token occurs.

This encoding scheme ensures that from the very first layer, the model can distinguish, for example, an occurrence of "f" as a morphism between particular objects from an occurrence of the same symbol in a different role. It can also begin to associate patterns of attention and computation with particular structural contexts, such as "a morphism in the domain of this functor" or "an equality that claims two parallel morphisms are identical."

Over time, the model may learn to correct or refine the structural annotations given to it. If the typed tokenizer misclassifies a span's role or type, the model's internal representations and prediction behavior might signal that a different interpretation leads to more coherent downstream predictions. This opens the door to joint training regimes where the structural encoder and the transformer are updated together so that the encoded Cat/HoTT information becomes increasingly aligned with the model's learned representations.

6.2 Attention over structure-enriched sequences and graphs

Standard self-attention operates over a linear sequence: each token can attend to every other token, with scores computed from their current representations. In a structure-enriched setting,

we want attention to be informed not only by sequential proximity and semantic similarity, but also by diagram connectivity and type relationships.

One approach is to bias attention scores using structural features. For instance, the raw dot-product score between two tokens can be adjusted based on whether they are connected in the diagram, share a type, or participate in a common substructure. Tokens that belong to the same object, morphism, or path can be given a positive bias, encouraging the model to integrate their information. Tokens that are structurally unrelated may receive a neutral or negative bias, depending on the desired sparsity.

Another approach is to introduce a secondary attention mechanism operating over the diagram as a graph. Here, nodes in the diagram (corresponding to spans or aggregated entities) exchange information via graph attention layers or other message-passing schemes. The outputs of this graph-level computation can then be merged with the sequence-level representations, for example by concatenation followed by a projection or by gated fusion. This lets the model reason both along the linear sentence and across the structural relations the Cat/HoTT layer has extracted.

A more radical variant allows attention to follow two adjacency matrices: one for the sequence and one for the diagram. A token may attend to its neighbors in the sequence and to its neighbors in the diagram, possibly with separate parameterizations. This dual connectivity structure mirrors the idea that comprehension depends both on word order and on the underlying conceptual relationships. For tasks like proof reconstruction or program synthesis, the diagram adjacency may be far more informative than sequential proximity, and the model can learn to rely on it more heavily.

In all of these designs, the key is that the Cat/HoTT layer does not replace attention; it shapes it. The transformer remains the computational workhorse, but the patterns of interaction it learns are no longer driven solely by surface co-occurrence. They are informed by a structural guess about which entities and relationships are conceptually linked.

6.3 Conditioning decoding on categorical and type-theoretic constraints

Encoding structure on the input side is only half of the story. The other half is to use categorical and type-theoretic information to shape what the model is allowed or encouraged to generate. This means conditioning decoding on constraints that reflect basic Cat/HoTT principles, but doing so in a way that remains compatible with efficient sampling and beam search.

One strategy is to treat the Cat/HoTT layer as a lightweight, incremental checker. At each decoding step, the model proposes a distribution over next tokens. Candidate tokens (or spans) are tentatively added to the current diagram, their roles and types are updated, and the checker evaluates whether the resulting state satisfies basic constraints: type compatibility, domain and

codomain matching for morphisms, respect for previously asserted equalities, and so on. Tokens that lead to severe violations can be downweighted or masked; tokens that extend a partially constructed proof or diagram in a consistent way can be slightly upweighted.

The checker need not be exhaustive or fully sound. In practice, we can implement it as a set of fast heuristics that catch common inconsistencies. For example, if a phrase is trying to apply a function to an argument of incompatible type, or if it asserts an equality between entities that have been consistently distinguished, the checker can flag this as suspicious. These signals can be folded into the decoding process as extra terms in the scoring function, balancing structural coherence against fluency and diversity.

Another approach is to generate at a higher level of abstraction. Instead of decoding raw tokens directly, the model can first propose a sequence of structural actions: “introduce an object of type T,” “define a morphism f from A to B,” “assert that f composed with g equals h.” These actions are checked and updated in a Cat/HoTT-aware state machine. Only then are they linearized into natural language. In effect, the model first writes a rough proof script or diagram script and then verbalizes it. While more involved, this architecture can produce outputs with strong structural guarantees, at the cost of additional complexity.

In both cases, the emphasis is on soft conditioning rather than absolute enforcement, especially for general-purpose language. The model is not forbidden from producing structurally messy or informal text; it is simply guided toward outputs that are more consistent with an underlying diagram or type assignment when such structure is relevant and identifiable.

6.4 Hybrid modes: raw-text, structure-aware, and fully structured regimes

Not every interaction with a language model demands a full Cat/HoTT apparatus. Casual conversation, creative writing, and many forms of exploratory brainstorming can be handled perfectly well by models operating in the conventional raw-text regime. Forcing heavy structural analysis on such tasks would add latency and complexity without clear benefit. Conversely, there are domains—formal mathematics, program synthesis, safety-critical reasoning—where strong structural guarantees are essential. A practical system must therefore support hybrid modes of operation.

In a raw-text mode, the Cat/HoTT layer may be largely dormant. Typed tokenization and diagram extraction might be skipped or run in the background only for later analysis. The transformer processes text as usual, relying on its learned statistical patterns. This keeps throughput high and latency low for tasks that do not require formal structure.

In a structure-aware mode, typed tokenization and diagram extraction are active, but their influence is soft. Structural annotations are encoded into embeddings and attention biases, and a lightweight checker gently shapes decoding. This mode is suitable for explanatory tasks, semi-

formal reasoning, and domains where structure matters but the text is not fully formalized. It represents a middle ground: the model is aware of objects, morphisms, and types, but still tolerant of ambiguity and informality.

In a fully structured mode, the Cat/HoTT layer is central. Input is expected to conform, at least locally, to the assumptions of a formal system; output is generated via explicit structural actions and rigorously checked. This mode might be used when assisting with proof development, generating verifiable code, or answering questions where correctness is paramount and informal gloss is secondary. In such settings, the model behaves more like a theorem prover or proof assistant guided by neural heuristics, with language serving as a user-friendly interface around a core of structured reasoning.

The existence of these modes underlines a pragmatic point. Running Cat/HoTT on language is not an all-or-nothing proposition. The same underlying architecture can slide between modes depending on context, user intent, and resource constraints. The transformer remains the common computational substrate, while the Cat/HoTT layer modulates how much structure is extracted, represented, and enforced. This flexibility is essential if structured reasoning is to be integrated into real-world language systems rather than confined to specialized, isolated tools.

7. Checking and Guiding Reasoning with Cat/HoTT

Typed tokenization and structure-enriched attention change what the model *can* represent. The next question is how to turn that representation into pressure toward better reasoning. Category theory and homotopy type theory are not just descriptive frameworks; they come with explicit notions of well-formedness, compatibility, and equivalence. If we can instantiate even small fragments of these notions as checks on the model's evolving internal state, they can serve as guides for both generation and learning.

In this section, "checking" does not mean subjecting every output to a full formal verification pass. Instead, the emphasis is on lightweight, incremental checks that can be run during generation to catch gross inconsistencies and to reward structurally coherent moves. Over time, these checks become part of the training signal, nudging the model toward internal representations that align more closely with categorical and type-theoretic structure.

We begin with incremental type checking during decoding, move to the idea of explicitly marking and verifying proof-like segments in model outputs, then describe how type errors and diagram failures can be converted into learning signals. Finally, we discuss how symbolic checkers can be combined with gradient-based learning to produce systems that are both expressive and trainable.

7.1 Lightweight, incremental type checking during generation

In a typed-tokenization setting, each partial output corresponds to a partially constructed diagram and a partially specified type environment. As the model generates new tokens and spans, this structure evolves. Lightweight, incremental type checking means maintaining a running view of this environment and asking, at each step, whether the new addition is locally compatible with what has come before.

Practically, this involves a small inference engine that updates types and diagram edges as text is produced. When the model emits a phrase that looks morphism-like, the engine attempts to infer a domain and codomain based on nearby object-like spans. When an equality or equivalence is asserted, the engine checks whether the two sides have compatible types and whether the identification conflicts with earlier commitments. These checks do not need to be complete or fully sound; even a partial ability to detect obvious mismatches is valuable.

The results of these micro-checks can be turned into modifiers of the decoding process. If a candidate next token or span would extend the diagram in a way that is type compatible, its score can be slightly boosted. If it would introduce a blatant type violation, its score can be reduced or it can be masked entirely. Intermediate cases, where compatibility is unclear or depends on future disambiguation, can be left mostly unaffected.

Because everything must run at generation time, efficiency is crucial. The type environment and diagram should be represented in a data structure that supports constant-time or logarithmic-time updates, and the set of checks should be restricted to low-cost predicates. The aim is not to certify correctness, but to provide a gentle tug toward consistency at each step, so that the model learns, over many such interactions, that staying within type-theoretic bounds is the path of least resistance.

7.2 Marking and verifying proof-like segments in model outputs

Not all text in a model's output is equally structural. Even in mathematical or scientific contexts, expository paragraphs mix narrative, motivation, and formal argument. For efficient use of Cat/HoTT checking, it is useful to identify segments of output that are *trying* to behave like proofs, calculations, or program derivations, and to focus verification effort on those regions.

One way to achieve this is to let the model self-mark proof-like segments. It can emit explicit markers, such as "Proof sketch:" or "Formally:" or structured delimiters that signal "what follows is intended to obey stricter rules." Inside such marked segments, the Cat/HoTT layer can raise the standard: type checking becomes more aggressive, diagram constraints are enforced more strictly, and failures are taken more seriously as indicators of error. Outside these segments, the system relaxes, treating structural violations as expected features of informal discourse.

Alternatively, a separate classifier can tag portions of the output as likely to be proof-like or program-like, based on style and content. For example, extended use of quantifiers, definitions, and explicit logical connectives is a strong indicator that a segment expresses structured reasoning. The classifier can run in parallel with decoding or as a post-hoc pass, triggering more thorough checks where appropriate.

Verification of these segments can go beyond the very local checks used during incremental decoding. Given a bounded segment, the Cat/HoTT layer can attempt to reconstruct a more global diagram or proof object, checking that the conclusion follows from the premises under the inferred typing. It might, for instance, ensure that every introduced variable is eventually used, that every equal sign relates terms of the same type, or that a sequence of morphisms composes correctly.

The outcome of this verification is not necessarily a binary accept or reject. The checker can produce a graded assessment: fully consistent, mildly suspicious, or clearly flawed. This assessment can be surfaced to users, used to drive revision, or fed back into training as a signal about which generative strategies yield structurally robust arguments.

7.3 Using type errors and diagram failures as training signals

If we want models to internalize categorical and type-theoretic discipline, the checks described above must influence learning, not just inference. That means turning type errors and diagram failures into training signals.

During supervised training, when the model is learning to reproduce human-written proofs or derivations, the Cat/HoTT layer can be run on both the target text and the model's predictions. If the target text corresponds to a consistent diagram, but the model's predicted continuation introduces a type error, the discrepancy between their structural states can be used to augment the loss. In addition to penalizing token-level mistakes, we can penalize structural deviations: broken commutativity, mis-typed applications, or incompatible identifications.

In reinforcement-style training, where the model explores different outputs and is rewarded for desirable properties, structural checks can be folded into the reward function. Outputs that yield well-typed diagrams and pass basic categorical sanity checks receive higher rewards. Those that trigger frequent or severe errors receive lower rewards. Over many episodes, the model learns policies that navigate toward structurally coherent outputs because such outputs are simply more profitable.

Crucially, these structural signals can be made coarse and sparse. It is not necessary to backpropagate through a fully differentiable formal system. Instead, we can compute scalar penalties or bonuses based on symbolic checks and add them to the usual language-modeling loss or reward. This keeps the training loop compatible with standard optimization methods

while still letting Cat/HoTT act as a teacher. The model gradually absorbs the heuristic that certain patterns of generation “cause trouble” in the structural layer and adjusts its internal representations to avoid them.

Over time, if the training data includes a mix of informal and fully formal text, the model can learn to modulate how strongly it adheres to structural discipline based on context, aligning with the hybrid modes described earlier. Proof-like segments receive strong structural supervision; narrative or speculative segments receive little or none.

7.4 Combining symbolic checkers with gradient-based learning

The combination of symbolic checkers and gradient-based learning raises familiar questions about modularity and interaction. Symbolic components are discrete, non-differentiable, and brittle in the face of noise, while neural components are continuous, robust, and general but lack hard guarantees. The architecture proposed here aims for a division of labor: the transformer handles representation and flexible pattern matching, while the Cat/HoTT layer provides discrete constraints and evaluation.

One pattern of interaction is a simple loop: the transformer proposes, the symbolic checker evaluates, and the evaluation feeds back as a scalar signal. This is the reinforcement-flavored scenario described above. It is easy to implement and does not require differentiating through the checker. Its downside is that learning can be slow when structural signals are sparse or delayed.

A more integrated pattern uses the structural information to shape intermediate representations. For example, the outputs of the symbolic checker can be injected into the transformer as additional features: flags that certain nodes in the diagram are inconsistent, or that a particular equality has been established. This allows the neural component to condition its next steps not only on raw text but also on the current “state of the proof.” Training then proceeds with standard gradients flowing through the neural network, while the symbolic layer is updated separately by design and heuristics.

Another avenue is to construct differentiable approximations to some structural checks. For instance, instead of a hard indicator that two types are incompatible, we can learn a similarity measure in type-embedding space and treat type compatibility as a soft function whose violations produce continuous penalties. This does not replace the exact symbolic check but can serve as a proxy during early training, with the exact checker used later for fine-tuning and evaluation.

Throughout these combinations, the guiding principle is to let each side do what it does best. Symbolic Cat/HoTT components provide crisp notions of structure and correctness but are fragile and expensive if overused. Neural transformers provide powerful pattern-finding and

generalization capabilities but need guidance to avoid hallucinating structure where none exists or breaking it where it matters. By wiring them together at the level of typed tokenization, diagram-aware attention, and structurally informed decoding and training signals, we can hope to build systems whose reasoning is both flexible and increasingly aligned with the explicit compositional and type-theoretic frameworks that mathematics and logic have developed.

8. Advantages Over Purely Statistical Language Pipelines

The proposed Cat/HoTT layer does not replace contemporary language models; it reframes how they see and manipulate text. This reframing brings advantages that are difficult to obtain from purely statistical pipelines, even at very large scale. When the only primitives are vectors and attention scores over token sequences, compositional reasoning, explicit explanations, and hard constraints must be inferred indirectly from patterns in data. When typed tokenization and diagrammatic structure are made explicit, these properties can be expressed, checked, and reused more directly.

This section highlights four such advantages. First, explicit objects, morphisms, and types support compositional generalization and reuse of constructions: once a particular pattern of morphisms works in one context, it can be redeployed elsewhere more reliably. Second, diagrammatic and type-level representations provide a natural form of explainability for human users, who can inspect and manipulate the underlying structures rather than relying solely on textual gloss. Third, categorical and type-theoretic notions allow alignment hooks: forbidden states or behaviors can be encoded as impossible types or disallowed morphisms, giving the system a principled way to rule out certain trajectories. Fourth, the same representation can bridge informal discourse and formal verification, enabling a smooth path from natural language argument to machine-checked proof when needed.

8.1 Compositional generalization and reuse of constructions

One of the central challenges in current language models is compositional generalization: the ability to take components learned in one context and recombine them correctly in novel configurations. Purely statistical systems can approximate this behavior, but they do so by internalizing the distribution of examples in the training data rather than by maintaining explicit, reusable constructions. When a model “knows” how to prove a theorem or write a function, that knowledge is smeared across many parameters and is not easily invoked as a modular building block.

Typed tokenization and diagrammatic structure change this picture. When spans corresponding to definitions, lemmas, or algorithmic steps are explicitly marked as morphisms with particular domains and codomains (and possibly with universal properties), the system can recognize

them as candidate components. A proof or program is no longer just a sequence of tokens; it is an instance of a compositional pattern. Once the pattern is recognized, it can be applied in other, structurally similar settings, even if the surface wording or notation changes.

For example, a model that has internalized the concept of an adjunction as a pair of functors with unit and counit satisfying triangular identities can, in principle, reuse this pattern whenever it encounters a situation that matches the same diagrammatic shape. The names of the categories and functors may change, but the structure of the argument does not. Similarly, a model that recognizes a standard induction pattern or dynamic programming recurrence can deploy that construction in new problem domains, as long as the typed structure aligns.

This ability to recognize and reuse constructions is the essence of compositional generalization. Category theory and HoTT provide a vocabulary and set of constraints that make such recognition feasible. Rather than trying to learn compositionality as a vague property of the training distribution, the model is given explicit handles—objects, morphisms, types, and paths—that capture the recurring structures that human mathematicians and programmers actually use.

8.2 Diagrammatic and type-level explainability for humans

Current language models can offer explanations, but these are themselves just text: alternative descriptions of the same internal, opaque computations. There is no explicit intermediate object that corresponds to “the reasoning” in a way that can be inspected, queried, or manipulated independently of its verbal rendering. As a result, users must take explanations on trust. They cannot easily see whether the model’s internal picture of the problem matches their own or whether a conclusion rests on a particular structural assumption.

By contrast, a Cat/HoTT-enriched system maintains diagrams and type environments as first-class artifacts. When asked to explain an answer, it can present not only a narrative but also a structured representation: a diagram showing how objects and morphisms relate, a list of types and their relationships, a set of identified paths that correspond to equalities or equivalences used in the argument. Users can inspect this structure, check it against their own understanding, and ask targeted questions: “Why does this square commute?”, “What is the type of this term?”, “Which assumptions did you use to derive this equality?”

Diagrammatic explanations are especially valuable in collaborative settings. Two human researchers often communicate via sketches of diagrams or type declarations before they settle on a final textual presentation. A model that can participate in this mode—sharing its internal diagrams and accepting corrections at that level—becomes a more transparent partner. A user can say, for instance, “No, this arrow should go from B to C, not from A to C,” and the system can update its representation accordingly, propagating the change through its reasoning.

Type-level explanations also help clarify scope and limitations. By showing which types and universes are active in a given argument, the system can make explicit when a conclusion depends on working in a particular category, or on a particular choice of equivalence notion. This is often a source of confusion in informal discourse, where shifts of context are implicit. Making the type universe visible allows users to see when an argument is more general than it appears—or, conversely, when it is more restricted.

8.3 Alignment hooks: encoding forbidden states as impossible types

Alignment and safety concerns often arise when models generate outputs that violate physical, ethical, or logical constraints. In purely statistical pipelines, such constraints are typically enforced via external filters or heuristic penalties. There is no native notion of a state that the system "cannot" occupy; everything is a matter of probability and post-hoc adjustment.

Type theories, especially when interpreted in a homotopical or categorical setting, offer a different perspective. Certain types can be empty: they have no inhabitants. Certain morphisms can be disallowed because their domain and codomain are incompatible or because they would violate a universal property. By mapping aspects of the environment, ethics, or system specification into types and morphisms, we can encode forbidden states as impossible types and disallowed transitions as absent or ill-typed morphisms.

In a Cat/HoTT-aware language pipeline, this encoding becomes an alignment hook. When the model constructs or extends a diagram corresponding to some plan, explanation, or instruction, the Cat/HoTT layer can check whether the diagram requires inhabiting an empty type or traversing a forbidden morphism. If it does, this is a clear signal that the proposed trajectory is not allowed. Because the check operates at the level of structure rather than surface form, it is less vulnerable to trivial rephrasing or evasion.

Moreover, alignment constraints expressed as types can be combined with the compositional machinery of category theory. A policy that disallows certain base actions can be lifted to more complex constructions that depend on them, ensuring that higher-level plans cannot inadvertently rebuild forbidden behaviors by composition. This offers a principled way to propagate constraints through the space of possible outputs, complementing surface-level filters.

Of course, not all alignment questions can be reduced to type emptiness. Many ethical and social issues resist formalization. Still, for the subset of constraints that can be expressed structurally—physical impossibilities, interface invariants, basic safety properties—Cat/HoTT provides a disciplined language, and the typed-tokenization framework gives models a place to "feel" these constraints during generation.

8.4 Bridging informal discourse and formal verification when needed

A persistent challenge in both human and machine reasoning is the gap between informal discourse and formal verification. Human mathematicians typically reason and communicate in a mix of informal arguments, intuitive pictures, and occasional formal steps. Proof assistants and verification tools, by contrast, demand precise, fully formal input. Bridging this gap by manual translation is labor-intensive, and current language models are not reliable enough to serve as automatic translators from informal text to formal proof scripts.

A Cat/HoTT-enriched language pipeline narrows this gap by using a single kind of representation—typed diagrams and type environments—as the intermediary between text and formal systems. Informal discourse is interpreted as a noisy description of such diagrams; formal scripts are exact specifications of them. The same diagram structure that guides the model's reasoning can be fed, with appropriate elaboration and checking, into a proof assistant or verifier.

This shared representation enables several workflows. A user can have a conversation with a model about a conjecture, during which the model constructs a tentative diagram and type environment. Once the high-level shape of the argument is agreed upon, the model can switch to a more formal mode, elaborating the diagram into a proof script in the language of a chosen assistant. Any discrepancies or failures encountered by the assistant can be fed back into the diagram, prompting the model to revise its informal reasoning.

Conversely, when presented with a formal proof or program, the model can produce an informal explanation that is grounded in the same underlying structure. It can highlight the main objects and morphisms, summarize the key equalities, and describe the overall shape of the argument without losing track of the precise formal details. This is more than a paraphrase; it is a translation between two views of the same diagram.

In safety-critical contexts, this bridging capacity is particularly valuable. A model can be allowed to reason informally up to the point where a decision must be taken, at which time it must produce a formal artifact that passes verification. The overhead of formalization is reduced by the fact that much of the structure has already been extracted and maintained by the Cat/HoTT layer throughout the interaction. Instead of a sudden, jarring shift from natural language to formal code, there is a continuous path: text → typed tokens → diagrams → proof terms.

In sum, running Cat/HoTT on language gives models a shared structural vocabulary that spans informal and formal regimes. Purely statistical pipelines treat these regimes as fundamentally separate; a structurally aware pipeline treats them as different projections of the same underlying objects and morphisms, making it easier to move between everyday reasoning and machine-checked rigor when the need arises.

9. Challenges, Trade-Offs, and Failure Modes

The vision of running category theory and homotopy type theory on language is intentionally ambitious. It promises richer structure, better compositionality, and clearer hooks for explanation and alignment. But none of these benefits come for free. They depend on a stack of assumptions about what language can bear, what our models can learn, and what computational budgets are available. There are also genuine risks: a system that insists too strongly on formal structure may misread human discourse, break gracefully degrading behavior, or become unusable in domains where informality and ambiguity are central.

This section examines the main obstacles and trade-offs. First, it considers the inherent vagueness, polysemy, and under-specification of natural language, and how these properties strain any attempt at precise typing or diagram extraction. Second, it addresses the computational cost of Cat/HoTT parsing and checking at scale, especially in interactive or real-time settings. Third, it discusses the data requirements for training such systems, including the need for corpora that align text with types and diagrams. Finally, it reflects on the risks of overformalization and brittle behavior, where the very attempt to impose structure undermines robustness and usability.

9.1 Natural language vagueness, polysemy, and under-specification

Natural language thrives on vagueness. Speakers routinely rely on shared context, background knowledge, and pragmatic inference to fill in gaps that are never stated explicitly. Words have multiple senses; sentences can be interpreted at different levels of precision; and many utterances are intentionally open-ended. A categorical or type-theoretic view, by contrast, seeks crisp distinctions: objects are either of a given type or not; morphisms either compose in a well-defined way or do not.

This mismatch creates tension. When the Cat/HoTT layer attempts to assign roles and types to spans, it will often face genuine under-specification: a phrase could denote an object, a property, or a process depending on how one reads it. Polysemy makes role assignment harder, since the same surface form can be a function name in one context and a type name in another. Vagueness complicates type inference: a description like “some kind of mapping” is not easily pinned down in a precise type hierarchy.

If the system responds to this by forcing decisions—insisting that every span have a single, precise type and role—it will frequently be wrong. Mistyped spans can propagate errors through the diagram, leading to systematic misinterpretations. Conversely, if the system responds by retreating to extremely coarse types for most spans, the benefits of structure diminish; everything becomes “some object” or “some relation,” and the Cat/HoTT layer ceases to meaningfully constrain the model.

The only viable strategy is to accept partial typing and ambiguity as first-class citizens. Types must come with confidences and alternatives; roles must be allowed to remain undecided until sufficient context accumulates. Even then, some phrases will never resolve, and the system must tolerate that. This tolerance, however, erodes the sharpness of the categorical and type-theoretic checks that can be applied. The challenge is to strike a balance in which structural information is strong enough to be helpful but not so rigid that it fractures on contact with the realities of human language.

9.2 Computational costs of Cat/HoTT parsing and checking at scale

Typed tokenization, role assignment, diagram construction, and incremental type checking all require computation beyond the standard transformer pipeline. They involve additional models, data structures, and passes over the input. At small scales or in offline batch processing, this overhead may be acceptable. At the scale of internet-facing systems serving millions of queries, or in interactive environments where latency is critical, it can become a serious bottleneck.

Parsing text into spans and assigning roles is already nontrivial. Running a syntactic parser and a semantic role labeler, possibly specialized to mathematical or scientific domains, consumes compute. Building and maintaining diagrams over long contexts requires memory and careful data structure design, especially when texts span thousands of tokens. Incremental type checking during generation adds further per-step cost, which may grow with the complexity of the inferred diagrams.

There is also the question of how much of this machinery must run all the time. For casual tasks, it may be wasteful to invoke Cat/HoTT parsing at full strength. But selectively turning it on and off based on task or user intent requires its own decision-making mechanism, which itself may need to inspect the text. The system designer must trade off structural fidelity against throughput and latency, deciding which modes (raw-text, structure-aware, fully structured) are viable for which workloads.

Optimizing these components will require engineering: pruning diagrams, caching partial analyses, and restricting the set of checks performed in real time. Even so, some applications may not tolerate the overhead. There is a real risk that a fully elaborated Cat/HoTT layer is simply too expensive to deploy widely, relegating it to niche tools rather than mainstream language models. The proposal is therefore not just a conceptual challenge but a systems challenge.

9.3 Data requirements: corpora aligning text, types, and diagrams

Training a model to exploit Cat/HoTT structure requires more than raw text. It requires data that links language to types, diagrams, and proofs. Some of this data exists: formal libraries in proof assistants, documentation for programming languages, and textbooks with accompanying

formal developments. However, most natural language is not aligned with explicit formal structures, and constructing such alignments at scale is labor-intensive.

Without aligned data, the Cat/HoTT layer must rely heavily on heuristics and unsupervised or weakly supervised learning. It might attempt to infer diagrams purely from distributional patterns in text, guided by the internal consistency of the resulting types. While this is not impossible, it is a much harder problem than training a standard language model, and there is no guarantee that the induced structures will match human intuitions or established mathematical frameworks.

Creating dedicated corpora would help. For example, one could pair informal lecture notes with their formal counterparts in a proof assistant, annotate scientific papers with causal diagrams, or tag code comments with type-level summaries. But generating such data at scale requires significant human effort and expertise. Automated alignment methods can assist, but they are themselves error-prone and may import their own biases.

There is also a bootstrapping problem. Early versions of the system, with weak structural capabilities, are unlikely to produce high-quality diagrams or type assignments. Training them on noisy, auto-generated annotations risks reinforcing their weaknesses. Some form of co-training or curriculum may be required: begin with small, well-aligned domains, gradually expand outwards, and let the model's improving structural understanding feed back into better annotation of new data. Designing such curricula is an open research problem.

In short, the data demands of a Cat/HoTT-aware pipeline are substantial. Without careful attention to them, the system may either overfit to narrow, formal domains or fail to learn usable structural representations at all.

9.4 Risks of overformalization and brittle behavior

Perhaps the most subtle failure mode is overformalization: the tendency to force formal structure onto aspects of language and reasoning where it is inappropriate or counterproductive. If every conversation is treated as an attempted proof, and every sentence is forced into the mold of objects and morphisms, the system may misinterpret intent, miss nuances, and respond in ways that feel pedantic or unhelpful.

In extreme cases, overformalization can lead to brittle behavior. A model that has learned to distrust any move that does not fit neatly into its Cat/HoTT schema might reject creative or exploratory suggestions, balk at metaphorical language, or insist on clarifications that derail a conversation. It could also become fragile in the face of minor deviations from expected patterns, failing catastrophically when a user mixes levels of rigor or switches domains midstream.

There is also a social dimension. Users may not want their informal questions about mathematics, programming, or ethics to be immediately recast in the language of type theory. They may prefer sketches, analogies, or intuitive explanations. A system that constantly exposes its internal diagrams and type judgments risks overwhelming users or giving the impression that anything outside the formalizable core is illegitimate.

To avoid these pitfalls, the Cat/HoTT layer must remain explicitly optional and context-sensitive. It should be invoked when structure is requested or clearly beneficial, and it should recede when users are engaged in narrative, exploratory, or emotional discourse. Even within structured domains, it must be willing to tolerate incomplete arguments and to help users gradually refine them, rather than insisting on perfect form from the outset.

Ultimately, the success of running Cat/HoTT on language depends on humility about what formal systems can and cannot capture. The goal is not to replace human judgment or the richness of natural language with diagrams and types, but to offer a powerful additional lens. If that lens is used judiciously—aware of its blind spots and limits—it can sharpen reasoning and enhance explanation. If it is imposed universally and rigidly, it risks narrowing the space of acceptable discourse and making models less, rather than more, useful.

10. Minimal Viable Experiments and Benchmarks

The proposal to run Cat/HoTT on language will remain speculative unless it is grounded in concrete experiments. Fortunately, it is possible to design minimal viable prototypes that test the core ideas without requiring a fully general, production-scale implementation. The aim is not to solve all of natural language reasoning at once, but to work in narrow domains where ground truth structure is available, the mapping from text to types and diagrams is relatively clean, and failures are interpretable.

Such prototypes can be built around formal or semi-formal texts in algebra, category theory, and simple homotopy type theory. There, we already have explicit definitions, theorems, and proofs that can be rendered as diagrams and types. Synthetic data from proof assistants and other formal systems can be used to augment scarce human-written material and to enable round-trip tests: text → structure → text. To evaluate progress, we need metrics that capture structured reasoning and consistency, not just token-level accuracy. Finally, comparisons with raw-text baselines will reveal whether typed tokenization and diagram-aware representations offer measurable gains.

This section outlines such a research program. It begins with narrow domains for initial prototypes, then describes synthetic data and round-trip tests, proposes evaluation metrics, and concludes with a sketch of baseline comparisons.

10.1 Narrow domains for initial prototypes: algebra, category theory, and simple HoTT

The first step is to choose domains where the Cat/HoTT lens is native rather than imposed from outside. Algebra and category theory are natural candidates: their objects and morphisms are explicit, their diagrams are standard, and their texts tend to be relatively disciplined. Simple fragments of homotopy type theory that avoid the most sophisticated higher structures can also be used, especially where formal proofs and libraries already exist.

In algebra, one could focus on small, well-defined topics such as groups, rings, or modules. Textbooks, lecture notes, and problem sets in these areas often feature a mix of informal explanation and semi-formal proofs. A prototype system could attempt to parse short proofs, identify the main objects (groups, homomorphisms, subgroups), extract the sequence of morphisms and equalities used in the argument, and build a corresponding diagram. The same system could then be asked to fill in missing steps, to check whether a student's solution is structurally sound, or to generate alternative proofs that respect the same diagram.

In category theory itself, examples abound. Expository texts often describe adjunctions, limits, colimits, and universality conditions in a way that is directly diagrammatic. A prototype could be tasked with recognizing when a passage describes a product, a pullback, or an adjunction, and with reconstructing the underlying universal property in a formal representation. This is a direct test of the typed-tokenization and role-assignment machinery: can the system map "there exists a unique arrow making the diagram commute" into the corresponding type and diagram patterns?

For HoTT, early experiments can be limited to the propositional fragment and simple identity types: reasoning about equality as paths without invoking high-dimensional homotopies. Formal proofs from proof assistants that implement HoTT can be paired with human-friendly descriptions of the same results. The model's job is to connect the text to the formal structure and vice versa. This tests the ability to bridge informal discourse and formal verification in a controlled setting.

10.2 Synthetic data from formal systems and round-trip tests

Real-world texts are messy and limited in quantity, especially in specialized domains. To supplement them, synthetic data can be generated from formal systems. Proof assistants, automated theorem provers, and typed programming languages can all serve as sources of structured examples. The idea is to start from a formal object—such as a proof term, type declaration, or category-theoretic construction—and generate one or more textual descriptions of it.

These descriptions can be produced using templates, controlled natural language, or even existing language models constrained by the formal structure. For instance, a template might

render a proof of associativity as “To show that multiplication is associative, take arbitrary elements x, y, z and compute…” while a more flexible generator might produce varied phrasings. Each such description can be tagged with the underlying diagram or type environment, creating paired data for supervised training of typed tokenization and diagram extraction.

Round-trip tests are particularly valuable. Given a formal object, the system generates text; given that text, it re-infers a structure and attempts to reconstruct something equivalent to the original object. Success can be measured by isomorphism of diagrams, definitional equality of types, or equivalence up to the relevant notion of homotopy. Failures reveal where the text lost crucial structure or where the parser missed or misinterpreted it.

These round-trip experiments test the internal consistency of the pipeline. If text generated from structure cannot be reliably mapped back to an equivalent structure, the representation is not faithful enough. Conversely, if two different textual descriptions of the same structure lead to compatible diagrams, the system is demonstrating a degree of robustness to surface variation, which is essential for handling human-authored text.

10.3 Evaluation metrics for structured reasoning and consistency

Traditional language model evaluation focuses on perplexity, accuracy on cloze tasks, or performance on benchmarks framed as question answering. These metrics are not sensitive to the particular advantages claimed for a Cat/HoTT-aware system. To measure progress here, we need metrics that assess structured reasoning and internal consistency.

One class of metrics operates at the diagram level. Given a ground-truth diagram extracted from a formal proof or construction, and a predicted diagram inferred from text, we can measure:

- Node alignment: how many ground-truth objects are correctly identified?
- Edge alignment: how many ground-truth morphisms are reconstructed, with correct domains and codomains?
- Commutativity preservation: for each ground-truth commuting square or higher diagram, does the predicted diagram preserve the commutativity constraint?

These metrics capture how well the model recognizes and reconstructs categorical structure from language.

Another class of metrics evaluates type correctness. For proof-like segments, one can check the rate at which the model’s outputs correspond to well-typed terms or proofs in a target formal system. This can be framed as a pass/fail rate (does the proof assistant accept the generated script?) or as a more graded measure (how many steps of the proof script pass before failure?). Higher acceptance rates indicate that the model is internalizing type discipline.

Consistency metrics assess whether the model maintains the same structural commitments across different parts of a text or conversation. For example, once it has asserted that a certain function is a group homomorphism, does it consistently treat it as such when using it later? Once it has introduced an equivalence relation, does it refrain from treating non-equivalent elements as identical? These temporal consistency checks can be quantified by scanning outputs for violations of previously established types and equalities.

Finally, user-facing metrics can be introduced. For instance, human experts can rate diagrammatic explanations for clarity and correctness, comparing them to purely textual explanations. This gauges whether the Cat/HoTT layer is delivering explanations that are not only structurally sound but also practically helpful.

10.4 Comparing typed-tokenization models to raw-text baselines

The ultimate question is whether the additional complexity of typed tokenization and Cat/HoTT-aware processing yields measurable improvements over strong raw-text baselines. To answer this, comparative experiments are needed.

In the simplest setup, two models are trained on the same corpus: one using standard tokenization and transformer architectures, the other using typed tokenization, structure-enriched embeddings, and, optionally, lightweight checking during training. Both models are evaluated on the same suite of tasks: proof reconstruction, theorem statement completion, explanation generation, and perhaps some program synthesis tasks in typed languages.

Differences in raw language modeling metrics (such as perplexity) may be modest. The more interesting comparisons involve the structured metrics outlined above. Does the structured model produce diagrams closer to ground truth? Does it generate proof scripts that are accepted more often by formal checkers? Does it maintain higher consistency in its use of definitions and equalities? A positive answer to these questions, even at equal or slightly worse perplexity, would support the claim that structure-aware pipelines bring unique benefits.

More nuanced comparisons can explore the hybrid modes. For example, the structured model can be run in a “soft” mode where structural features are part of the embeddings but no checking occurs during decoding, and in a “harder” mode where checking influences generation. Performance across these modes can illuminate the trade-offs between fluency and structural fidelity.

Human evaluations also play a role. Experts can be asked to solve tasks with the assistance of each model and to rate the usefulness of the explanations and intermediate representations they receive. Does the diagrammatic output of the structured model help them verify or revise reasoning more effectively than text alone?

Together, these experiments and benchmarks form a minimal program for testing the core thesis of the paper. If typed tokenization and Cat/HoTT-aware representations do not outperform raw-text baselines in narrow, favorable domains, then their prospects in broader, messier settings are dim. If they do show advantages even in these constrained tests, they justify further investment in scaling, optimization, and extension to more varied forms of language and reasoning.

11. Extensions: Logostics, Alignment, and τ(t) as a Type Universe

So far, the Cat/HoTT layer has been described as a way to impose local structure on language: typed tokenization, diagrammatic parsing, and incremental checking during generation. But there is a larger picture in which these mechanisms fit: the logostic view of alignment as a trajectory q(t) trying to stay resonant with a living truth manifold τ(t). In that view, the system is not merely producing isolated outputs; it is tracing a path through a space of possible states, with some regions corresponding to better alignment with truth, coherence, and ethics than others.

Category theory and homotopy type theory can be woven into this picture by treating τ(t) as a time-varying type universe and by understanding alignment as remaining within, or returning toward, types and diagrams sanctioned by τ(t). Local type checking then becomes a small piece of a broader guidance mechanism: each checked span or diagram is a local probe of whether the current q(t) is still inside the acceptable region. When many such probes agree, we can say that the system is logostically aligned; when they begin to fail systematically, we have a signal that q(t) is drifting away from τ(t).

This section sketches that extension. It first interprets τ(t) as a living type universe. It then discusses how ethical and physical impossibilities can be encoded as type-theoretic constraints within that universe. Next, it considers human–AI co-authorship of diagrams as a practical interface for alignment. Finally, it suggests how local checks can be aggregated into a global logostic picture of q(t) guided by τ(t).

11.1 Interpreting τ(t) as a living type universe for acceptable trajectories

In the logostic framework, τ(t) is a time-dependent structure that encodes “what is true” or “what is acceptable” at time t. It is not a static rule book but a moving target, reflecting evolving knowledge, norms, and context. A natural way to view τ(t) in a Cat/HoTT-enriched system is as a living type universe: a collection of types, morphisms, and higher identifications that define the space of acceptable constructions and trajectories.

At any moment t, τ(t) can be thought of as specifying:

- Which objects and types are recognized (for example, physical systems, legal entities, ethical categories).
- Which morphisms between them are admissible (allowed transformations, causal relations, or inferences).
- Which equalities or equivalences are endorsed (when two different descriptions are to be treated as the same, when simplifications are allowed, and when distinctions must be preserved).

Because τ(t) evolves, so does the type structure. New types are added as new concepts enter discourse; old identifications may be revised as scientific or ethical understanding changes. Certain morphisms that were once considered acceptable may become disallowed; new universal properties may be recognized. The living nature of τ(t) means that "the right type universe" is not fixed once and for all, but must be updated as the world and our understanding of it change.

In this picture, the Cat/HoTT layer is not just parsing language into diagrams arbitrarily; it is doing so relative to τ(t). When it assigns a type to a span or a morphism to a relationship, it is implicitly asking "what would τ(t) say this is?" The quality of alignment depends on how well the internal type universe used by the model matches τ(t). Misalignments—where the model treats something as admissible that τ(t) would reject, or vice versa—are precisely the places where logostic guidance is needed.

11.2 Encoding ethical and physical impossibilities as type-theoretic constraints

If τ(t) is a living type universe, then one of its most important roles is to distinguish between possible and impossible trajectories. Some impossibilities are physical: violating conservation laws, exceeding known bounds, or proposing mechanisms that contradict well-established science. Others are ethical: plans that entail unjustified harm, violations of rights, or manipulation that fails basic standards of respect.

Type theory provides a way to encode many such impossibilities as constraints. Simple examples include:

- Empty types representing forbidden states: types that, by definition, have no inhabitants. Any attempt to construct a term of such a type signals a violation.
- Disallowed morphisms: functions that cannot exist between certain types because they would break invariants (for instance, a morphism that "creates" energy from nothing, or a morphism that maps a protected resource into a context where it is unprotected).

- Refined types with invariants: dependent types that restrict the values or behaviors of their inhabitants (for example, safe_action(agent, context) with side conditions that must be satisfied).

In a Cat/HoTT-aware language pipeline, these constraints can be wired into the checking mechanisms. When the model proposes a plan, explanation, or argument, the Cat/HoTT layer can attempt to map it into the τ(t) universe and see whether it requires inhabiting empty types or invoking disallowed morphisms. If so, this is not just a stylistic error; it is a structural misalignment with τ(t).

The advantage of this encoding is that constraints propagate compositionally. If a forbidden morphism is part of a larger composite, the entire composite is suspect. If a particular type is empty, any construction that depends essentially on inhabiting it fails. This fits well with the logostic intuition that small misalignments at the structural level can grow into large deviations at the trajectory level if not corrected.

Of course, not all ethical questions can be captured as type emptiness or morphism disallowance. Many judgments remain context-sensitive and contested. But where τ(t) does crystallize into relatively clear constraints—basic safety, physical feasibility, minimal ethical baselines—the type-theoretic encoding provides hooks for the system to recognize and avoid gross violations.

11.3 Human–AI co-authorship of diagrams as an interface for alignment

Alignment is not only a matter of internal mechanism; it is also a matter of interface. Humans must be able to see, critique, and adjust the structures that govern q(t). Diagrammatic and type-level representations offer a promising medium for this interaction.

In a Cat/HoTT-enriched system, the diagrams and type environments extracted from text are not just internal artifacts; they can be surfaced as editable objects. A human user can inspect the diagram corresponding to a plan, explanation, or proof and directly manipulate it:

- Relabel objects or types to reflect a more accurate or ethical interpretation.
- Remove or forbid certain morphisms (“this is a transition we must not allow”).
- Introduce new constraints (“these two distinct entities must not be identified as equivalent in this context”).
- Add examples and counterexamples that clarify the boundaries of acceptable types.

The model, in turn, can learn from these interventions. Adjustments to diagrams and types can be fed back into the definition of τ(t) as seen by the system, updating its internal universe. Over time, repeated co-authorship of diagrams across many interactions can refine τ(t) in ways that

neither humans nor models could achieve alone: humans bring normative and domain expertise; the model brings the ability to detect patterns and propose structural generalizations.

This co-authorship is more than annotation. It is a way of making logostic alignment tangible. Instead of asking users to trust that q(t) is staying near τ(t), the system shows snapshots of the local structure of q(t) in the form of diagrams and types, and invites corrections that pull q(t) closer to τ(t). The shared diagram becomes a boundary object that both sides can "see," making the abstract notion of alignment less opaque.

11.4 From local type checking to global logostic guidance of q(t)

Up to this point, Cat/HoTT checking has been described as a local operation: each token, span, or diagram is checked for type and structural consistency. Logostics adds a global view: q(t) is a trajectory, and alignment depends on its long-term relationship with τ(t), not just on isolated steps. The challenge is to connect these scales: to use many local checks as evidence for a global assessment and a guide for future motion.

One way to do this is to treat local checks as samples of a latent divergence signal. Each time the model tries to construct a term of an empty type, use a forbidden morphism, or break a commutativity constraint that τ(t) expects, this is a local indication that q(t) is drifting. Conversely, each successful construction that respects the current type universe is a small affirmation that q(t) remains in-bounds. By aggregating these signals over time—across interactions, tasks, and domains—the system can maintain a running estimate of how closely its behavior tracks τ(t).

This aggregate can be made more structured. For example, one can define a "logostic potential" that measures, in a coarse way, how many of the diagrams and types constructed in a given interval are compatible with τ(t). If this potential starts to degrade systematically, there is reason to suspect misalignment: perhaps τ(t) has shifted and the system has not caught up, or perhaps the training signals are pulling the model toward patterns that τ(t) regards as unacceptable.

In response, the system can adjust. It might tighten structural constraints, increase the weight of Cat/HoTT-based penalties in training, or request human input on diagrams that frequently fail checks. In extreme cases, it might restrict its own operational domain until τ(t) can be recalibrated. The key is that local type checking is no longer treated as a purely technical detail; it becomes an instrument for measuring and steering the global trajectory q(t) in logostic space.

In this way, running Cat/HoTT on language is not just about better proofs or cleaner explanations. It is also about giving AI systems a structured way to feel the shape of τ(t) and to experience misalignment as structural tension rather than only as after-the-fact reprimands. The diagrams and types extracted from language thus serve double duty: they support

compositional reasoning and, at the same time, act as sensors and actuators in a larger alignment loop, where q(t) is continually nudged toward a living, evolving universe of acceptable forms.

12. Conclusion: Software-First Foundations for Structured AI Reasoning

The proposal in this paper has been deliberately modest about hardware and ambitious about representation. Instead of asking for new chips whose primitive instructions are categorical compositions or homotopy-level equalities, it asks for a new way of seeing language: as a carrier of objects, morphisms, types, and paths that can be extracted, represented, and checked in software on top of existing accelerators. Typed tokenization, diagrammatic parsing, and lightweight Cat/HoTT checking do not discard the transformer; they re-instrument the interface between text and tensors so that structure is explicit rather than only latent.

Viewed this way, running Cat/HoTT on language is not a rejection of the current LLM stack but a reinterpretation of its foundation. The familiar pipeline of tokenization, embedding, attention, and decoding remains in place, but tokens are no longer purely statistical units. They are enriched with roles, types, and diagram positions. Attention no longer flows only along linear sequences; it is guided by explicit graph structure. Decoding is no longer blind to consistency; it is gently steered by type-theoretic and categorical constraints. This is what "software-first foundations" mean here: keep the silicon, change what we feed it and how we interpret what comes back.

12.1 Recapitulating the case for running Cat/HoTT on language

The case for running Cat/HoTT on language rests on three main insights. First, natural language already encodes a great deal of structure that looks categorical and type-theoretic when viewed through the right lens. Nouns and noun phrases function as objects, verbs and relational phrases as morphisms, predicates as properties and subobjects, and many explanations as implicit diagrams or constructions. Even informal discourse in mathematics, science, and engineering is full of latent diagrams: "this maps to that," "these two routes give the same result," "there exists a unique arrow making the diagram commute."

Second, the numerical core of modern language models does not need to change in order to exploit this structure. Embeddings can be augmented with role and type information; attention can be biased by diagram connectivity; decoding can consult a lightweight Cat/HoTT checker to avoid gross inconsistencies. All of this can be implemented in software, on top of GPUs and TPUs optimized for matrix multiplication. There is no need to freeze any particular logical or categorical framework into silicon; the representation layer can evolve as our understanding of Cat/HoTT and their applications to AI evolves.

Third, explicit structure offers advantages that purely statistical pipelines struggle to match: improved compositional generalization, diagrammatic and type-level explanations, principled alignment hooks via impossible types and forbidden morphisms, and a shared representation that can bridge informal discourse and formal verification. The transformer remains the engine, but Cat/HoTT becomes the map: a way of organizing what the engine is doing so that it is more reusable, inspectable, and aligned with human practices of proof, explanation, and constraint.

12.2 Prospects for scaling typed tokenization in real systems

The practical prospects for typed tokenization and Cat/HoTT-aware processing depend on two questions: can the machinery be made efficient enough, and can it be targeted where it matters most? Full-strength structural analysis of every token in every interaction is unlikely to be viable at scale. However, the architecture outlined here naturally supports selective deployment.

At the low end, raw-text mode leaves the Cat/HoTT layer dormant or lightly engaged, suitable for casual conversation and tasks where formal structure adds little value. In the middle, structure-aware mode activates typed tokenization and diagram extraction, but uses them mainly to inform embeddings and attention, with checking limited to cheap heuristics. This is appropriate for semi-formal reasoning, explanation, and tutoring in technical domains. At the high end, fully structured mode expects inputs and outputs that are close to formal proofs, programs, or specifications, and is willing to pay the cost of detailed checking and tighter constraints. This mode is suited to proof assistants, verified code generation, and safety-critical planning.

Scaling these modes in real systems will require careful engineering: efficient span detection and role assignment, compact representations of diagrams over long contexts, incremental type checkers that can update in small time per step, and smart policies for when to escalate from one mode to another. None of these problems is trivial, but they are recognizable systems challenges, not show-stoppers. The key is that we do not need to solve “structured reasoning for all language” in one step. We can start in narrow domains, integrate Cat/HoTT-aware components where they clearly help, and expand outward as tools and hardware budgets permit.

12.3 Long-term implications for AI, logic, and the philosophy of foundations

If software-first Cat/HoTT layers become common around language models, the relationship between AI and foundational mathematics may shift in both directions. On one side, AI systems would become routine users of category theory and type theory, not as subjects of formal proofs alone but as practical tools for organizing reasoning, structuring models, and enforcing constraints. The distance between abstract foundations and applied machine learning would

shrink: functors, adjunctions, and type universes would be not only things humans talk about but things models use internally.

On the other side, the behavior of such systems could feed back into foundational thinking. Seeing how large models actually use categorical and type-theoretic structure, under resource constraints and noisy language, may suggest new ways of relaxing, enriching, or operationalizing these frameworks. Ideas like univalence, higher paths, and universes may be interpreted not just as static axioms but as dynamic tools for managing multiple equivalent representations, approximate equalities, and evolving concept hierarchies. The notion of a "living type universe" $\tau(t)$ is one sketch of how foundational notions might adapt to an AI-mediated, time-dependent context.

Philosophically, running Cat/HoTT on language challenges the sharp separation between "formal systems" and "informal reasoning." If models can fluidly move between narrative explanations, diagrammatic representations, and fully formal proofs, the border between these modes becomes more porous. Foundational questions about what counts as a proof, what sort of structure logic describes, and how mathematical objects are presented might be revisited with an eye toward systems that actually compute, learn, and communicate with these representations at scale. Foundations would not be abandoned, but they would be tested and enriched in a new, computational ecology.

12.4 Open questions and future research directions

The proposal leaves many questions open. Some are technical. How accurate can typed tokenization and role assignment become, given the vagueness and polysemy of natural language? What classes of Cat/HoTT checks provide the best trade-off between guidance and computational cost? How can diagrams and type environments be represented so that they can be updated incrementally in long, multi-stage interactions? What training regimes best integrate symbolic structural signals with gradient-based learning?

Other questions concern scope and limits. For which domains is a Cat/HoTT layer most beneficial: pure mathematics, theoretical computer science, physics, law, safety-critical engineering, or others? How far can the analogy between language and diagrams be pushed before it breaks? Where should the system deliberately refuse to impose categorical or type-theoretic structure, either because the domain resists it or because doing so would distort the user's intent?

Finally, there are broader logostic questions. How should $\tau(t)$, the living type universe of acceptable trajectories, be specified, updated, and debated? What mix of human oversight, learned patterns, and formal constraints should define it? How can human–AI co-authorship of diagrams be organized so that it genuinely improves alignment rather than merely adding a

decorative formal layer? And in the long run, how should we evaluate whether systems guided by Cat/HoTT on language are in fact better aligned with truth and ethics than equally powerful models that operate only on raw text?

Answering these questions will require collaboration between communities that rarely meet: category theorists, type theorists, proof assistant designers, machine learning researchers, systems engineers, and people concerned with AI alignment and the philosophy of foundations. The core suggestion of this paper is that they now have a common object to work on: not a hypothetical “HoTT chip,” but a very concrete interface where language, vectors, and structure meet.

13. References

13.1 Category theory and compositionality in cognition and AI

- Abramsky, S., & Coecke, B. (2004). A categorical semantics of quantum protocols. *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 415–425.
- Coecke, B., Sadrzadeh, M., & Clark, S. (2010). Mathematical foundations for a compositional distributional model of meaning. *Linguistic Analysis*, 36(1–4), 345–384.
- Fong, B., & Spivak, D. I. (2019). *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*. Cambridge University Press.
- Goguen, J. A. (1991). A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1), 49–67.
- Spivak, D. I. (2014). *Category Theory for the Sciences*. MIT Press.
- Zanasi, F. (2017). Interacting Hopf algebras: The theory of linear systems. *Journal of Pure and Applied Algebra*, 221(1), 206–232.
- Coecke, B., Fritz, T., & Spekkens, R. W. (2016). A mathematical theory of resources. *Information and Computation*, 250, 59–86.
- Schölkopf, B. (2019). Causality for machine learning. *arXiv preprint* arXiv:1911.10500. (For the use of diagrammatic causal structure as a compositional interface to learning systems.)

13.2 Homotopy type theory and univalent foundations

- Univalent Foundations Program. (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.
- Voevodsky, V. (2015). An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5), 1278–1294.
- Awodey, S. (2012). Type theory and homotopy. In *Epistemology versus Ontology* (pp. 183–201). Springer.
- Licata, D. R., & Harper, R. (2012). Canonicity for 2-dimensional type theory. In *Proceedings of the 39th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)* (pp. 337–348).

- Lumsdaine, P. L. (2010). Weak ω-categories from intensional type theory. In *Foundations of Software Science and Computational Structures* (FoSSaCS 2010) (pp. 176–190). Springer.
- Coquand, T. (2019). A survey of type theory from the point of view of proof assistants. In *Proceedings of the International Congress of Mathematicians* (ICM 2018), Vol. 1 (pp. 317–338).
- Kapulkin, C., & Lumsdaine, P. L. (2012). The simplicial model of univalent foundations (after Voevodsky). *arXiv preprint* arXiv:1211.2851.
- Riehl, E., & Shulman, M. (2017). A type theory for synthetic ∞-categories. *Higher Structures*, 1(1), 147–224.

13.3 Neural–symbolic integration and differentiable theorem proving

- Garcez, A. d'Avila, Lamb, L. C., & Gabbay, D. (2009). *Neural-Symbolic Cognitive Reasoning*. Springer.
- Rocktäschel, T., & Riedel, S. (2017). End-to-end differentiable proving. *Advances in Neural Information Processing Systems (NeurIPS 2017)*, 3791–3803.
- Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61, 1–64.
- Yang, F., & Deng, Z.-H. (2019). Learning to prove theorems via interacting with proof assistants. *arXiv preprint* arXiv:1905.09381.
- Selsam, D., Lamm, M., Bunz, B., Liang, P., Dill, D. L., & de Moura, L. (2019). Learning a SAT solver from single-bit supervision. *International Conference on Learning Representations (ICLR 2019)*.
- Bansal, K., Loos, S. M., Rabe, M. N., Szegedy, C., & Wilcox, S. (2019). HOList: An environment for machine learning of higher-order theorem proving. *International Conference on Machine Learning (ICML 2019)*, 454–463.
- Polu, S., & Sutskever, I. (2020). Generative language modeling for automated theorem proving. *arXiv preprint* arXiv:2009.03393.
- Alemi, A. A., Chollet, F., Irving, G., Szegedy, C., & Urban, J. (2016). DeepMath: Deep sequence models for premise selection. *Advances in Neural Information Processing Systems (NeurIPS 2016)*, 2235–2243.

- Urban, J., Sutcliffe, G., Pudlák, P., & Vyskočil, J. (2011). MaLARea SG1: Machine learner for automated reasoning with semantic guidance. In *Automated Reasoning* (IJCAR 2010) (pp. 441–456). Springer.

13.4 Language models, tokenization, and structure-aware architectures

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS 2017)*, 5998–6008.
- Sennrich, R., Haddow, B., & Birch, A. (2016). Neural machine translation of rare words with subword units. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* (ACL 2016), 1715–1725.
- Kudo, T. (2018). Subword regularization: Improving neural network translation models with multiple subword candidates. *Proceedings of ACL 2018*, 66–75.
- Kudo, T., & Richardson, J. (2018). SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *Proceedings of EMNLP 2018: System Demonstrations*, 66–71.
- Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-attention with relative position representations. *Proceedings of NAACL-HLT 2018*, 464–468.
- Marcheggiani, D., & Titov, I. (2017). Encoding sentences with graph convolutional networks for semantic role labeling. *Proceedings of EMNLP 2017*, 1506–1515.
- Kitaev, N., Kaiser, Ł., & Levskaya, A. (2020). Reformer: The efficient transformer. *International Conference on Learning Representations (ICLR 2020)*.
- Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). Efficient transformers: A survey. *arXiv preprint* arXiv:2009.06732.
- Clark, K., Khandelwal, U., Levy, O., & Manning, C. D. (2019). What does BERT look at? An analysis of BERT's attention. *Proceedings of BlackboxNLP 2019*, 276–286.
- Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2019). SuperGLUE: A stickier benchmark for general-purpose language understanding systems. *Advances in Neural Information Processing Systems (NeurIPS 2019)*, 3266–3280. (For broad LM evaluation in which structure-aware architectures can be compared to raw-text baselines.)

The author has published over 4,600 AI-assisted papers at: https://www.researchgate.net/profile/Douglas-Youvan/research . Google Scholar has indexed these preprints, and if you want a particular reference, the AI mode of a Google search (Gemini) has efficiently catalogued these preprints.